

React Native 跨平台移动应用开发

阙喜涛 编著

電子工業出版社

Publishing House of Electronics Industry

北京 · BEIJING

内 容 简 介

React Native 是 Facebook 公司最新推出的，强大的、开源的跨平台移动应用开发框架，它能大幅减少跨平台移动应用开发的工作量，并且代码结构清晰、简单易懂。同时 React Native 框架采用模块化结构，使应用版本的更新迭代非常简单。随着它的日趋成熟，React Native 必然会成为移动应用开发的主流技术。

本书由浅入深、系统地介绍了使用 React Native 框架跨平台开发所需要用到的知识。本书每一章都专注于阐述某一方面的知识，配合若干个原创的、精小的例程，能让读者系统、快速地掌握该方面知识。

本书是按照有基本的编程基础知识，刚学习 JavaScript 基本语法的读者最佳学习路径来编写的。有一定基础的开发人员也可以将本书作为 React Native 开发的“字典”来使用，在开发时可以快速查找相关知识点的细节。

希望通过本书，能将最新的 React Native 开发技术介绍给国内广大开发者，让国内移动应用开发进入一个新的时代，让开发者用比较少的精力就能高效、美观地完成移动应用开发。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

React Native 跨平台移动应用开发 / 阙喜涛编著. —北京：电子工业出版社，2016.6
ISBN 978-7-121-28707-7

I. ①R… II. ①阙… III. ①移动终端—应用程序—程序设计 IV. ①TN929.53

中国版本图书馆 CIP 数据核字（2016）第 094122 号

策划编辑：孙学瑛

责任编辑：葛 娜

印 刷：

装 订：

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱

邮编：100036

开 本：787×1092 1/16 印张：22.75

字数：596 千字

版 次：2016 年 6 月第 1 版

印 次：2016 年 6 月第 1 次印刷

定 价：79.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888

质量投诉请发邮件至 zltts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819 faq@phei.com.cn。

前言

我对移动应用开发大势的粗浅理解

我必须向大家坦白：我现在是个没有良心的人。因为写这本书不仅把我的良心用完了，还透支了很多。本书所有的示例代码都是原创，并且在代码旁有注释解说，绝没有扔一大段代码给读者自己慢慢看的情况。

仔细想来，所有主流手机平台应用开发我都涉足过，它们是 J2ME、Symbian、Series 60 开发平台、Windows Mobile、Android、iOS、Windows Phone。许多年前我写过一本 Series 60 平台移动应用开发的书。当我写到一大半时，Symbian 手机操作系统、Nokia 手机、Series 60 开发平台这片森林从地球表面消失了……

如果问这么多年我对移动应用开发最深的感触是什么，那就是移动应用开发太特么的难了！难就难在应用需要能运行在不同的手机上。开发者的代码要适配不同的手机操作系统（这意味着要使用不同的编程语言编写）、不同的手机硬件能力（比如开发者的应用需要使用 GPS 定位，然后发现某型号手机没有 GPS 定位功能）、不同的手机屏幕分辨率（想象一下当开发者发现应用程序在另一种屏幕分辨率下显示得乱七八糟时的惊喜）。

随着科技的发展，手机开发也在向好的方向不停地转变。Android 与 iOS 两大手机操作系统称霸江湖（这种稳定对开发者是一件好事儿，相互竞争也让两大操作系统都在不断地提升完善）。手机硬件配置越来越强大，能力越来越强大，价格越来越平易近人。手机现在已经成为了普通老百姓无时无刻不随身携带的电脑。伴随着这种趋势，市场对移动应用开发的需求也越来越多，并且要求越来越高。

这么多年来，移动应用开发者做梦都希望有一个能跨平台的开发工具，让他们不要把同一个移动应用使用不同的开发语言写两遍（或者三四遍）。但跨平台开发工具的实现很难，很多先驱者倒在了这条路上。直到 Facebook 给大家带来了 React Native。

让我以一个预言结束我的唠叨吧。在西方传说故事里那些没良心的巫师的预言忒准了，所以请读者对我的预言稍微有些信心。嗯，我的水晶球呢？啊，在这里。（装模作样好长一段时间）预言：我看到了一个新的伏地魔正在崛起，他会统治这个世界，他那高高的巫师帽上 React Native

这个名字不时地散发着邪恶的光芒！

写作本书的目的

React Native 项目代码是开源的，项目代码中的例程也是开源的，项目文档也是在网上公开的，所有人都可以非常方便地获取浏览。但目前 React Native 的文档假设它的读者有相当深厚的开发方面的知识与经验，并且有 React.js 开发基础。也就是说，React Native 的公开文档对初学者并不是非常友好的。React Native 项目中的例程代码对初学者而言同样有些高深。在各个 React Native 论坛上，React Native 的文档不够好是大家的共识。

笔者非常努力地把这本书写好，让它从简入深，通俗易懂。本书绝对不是粗制滥造、简单翻译的书籍。书中讲述的知识点结构、例程设计都倾注了笔者大量的心血，与网上公开文档的差别非常大。本书对读者最大的价值是：把读者通过阅读网上公开文档、项目例程学习 RN 开发技术所需要的 800 小时左右的时间（视个人基础有所不同）缩短为 300 小时左右。

本书中的 6.5 节、第 11 章、14.5 节都是笔者精心整理调研出来的技术，在官方文档中没有提及，希望能让更多的开发者享受到 React Native 开发的便利性。

希望通过本书，能将 React Native 开发技术介绍给国内广大开发者，让国内移动应用开发进入一个新的时代，让开发者用比较少的精力就能高效、美观地完成移动应用开发。

读者对象

本书的读者对象如下：

- Android 平台移动应用研发工程师
- iOS 平台移动应用研发工程师
- JavaScript 软件研发工程师
- 计算机相关专业的学生

如何阅读本书

React、React.js、React Native

对于初学者常见的困惑是弄不清 React、React.js、React Native 三者之间的关系。这是情有可原的。初学者经常发现在很多讨论 React Native 技术的资料中，怎么突然说到 React.js 上去了？过一会儿怎么又好像说到 React 基础框架上去了？因此在讨论如何阅读本书之前，有必要先说明一下这三者的关系。React 是基础框架，是一套基础设计实现理念，开发者不能直接使用它来开发移动应用或者网页。在它之上发展出了 React.js 框架用来开发网页，发展出来 React Native 用来开发移动应用。因为 React 基础框架与 React.js 框架是同时出现、同时进化发展的，这就造成了 React 基础框架的基本概念、设计思想都是在 React.js 的相关文档中描述的。后来，Facebook 推出 React

Native 后，也没有把 React 的相关概念文档从 React.js 文档中分离出来。这就导致出现了学 React Native 要去看 React.js 文档，说 React Native 不时会说到 React.js 的情况。如果开发者只想关注移动应用开发，那么在 React.js 的如何开发网页的文档中整理出来 React 基础框架知识是个不轻松的活儿。现在，本书推出了，有了这本书，读者可以不用再去查看 React.js 文档了。

预备知识

学习 React Native 开发需要基础的 JavaScript 编程知识。我估计有一部分读者可以在某个手机平台使用该平台原生语言进行移动应用开发，但对 JavaScript 只是有所耳闻。如果是这种情况，不用担心。读者只需要随便找一本 JavaScript 入门级的书籍（甚至是网上的教学性网页），阅读关于基础语法的章节，花上一天时间（包括找资料的时间）学习相关知识点，接下来就可以通过本书学习 React Native 开发了。所需要的知识点有：

（1）JavaScript 语法。包括语句、注释、变量、数据类型、数组（注意关联数组，Java、C++ 的数组中没有这个概念）、对象的基本知识。

（2）操作运算符。这个与 Java、C++ 基本上是一样的，读者快速过一下就行了。

（3）条件语句、循环语句、switch 语句。这个与 Java、C++ 基本上是一样的，读者快速过一下就行了。

（4）函数。JavaScript 中函数也是一种变量，知道了这一点，其他与 Java、C++ 基本上是一样的，读者快速过一下就行了。

（5）对象。JavaScript 的对象定义、实现比 Java、C++ 宽松很多，读者得稍微适应一下。

这些基本的知识点，大多与 Java 或者 Objective-C 的相关知识点很类似。如果读者有基础，阅读这些知识点最多只需要一天的时间（半天看完也不是难事儿）。读者不需要搭建 JavaScript 的开发环境来练习、巩固这些知识点。因为在 React Native 的开发环境中编写代码就可以练习这些基本的 JavaScript 知识点，在 React Native 学习中就会巩固这些 JavaScript 基本知识。

React Native 在开发中用到了其他 JavaScript 的高级知识点与 ES 6 的一些新特性。但读者不需要马上去学习这些内容。在通过本书学习 React Native 开发过程中需要使用到的 JavaScript 高级知识点时，会指出在附录 A 的什么位置讲解了这些高级知识点，便于读者快速查看。附录 A 不是 JavaScript 知识点的全面讲解，只是让读者对 React Native 开发中需要使用到的 JavaScript 知识点的理解足够进行 React Native 开发。

相关下载地址

笔者希望读者在阅读本书时，能在理解的基础上将例程代码输入到电脑中。输入的过程是一个消化吸收的过程。输入完成后，运行代码，并且按照提示或者针对自己有疑虑的地方进行修改，以便深入理解各个知识点。

正因为如此，本书前面章节中那些短小精悍的例程并没有附在一张光盘上，或者在网上提供

下载地址，而是需要读者自己手动输入电脑。

本书还有一些不需要读者手动输入的代码，笔者在 GitHub 上提供了一个网址供读者自行下载。网址是：<https://github.com/xिताoque>。

本书结构

本书讨论的 React Native 开发特性覆盖了 2016 年 3 月 2 日发布的 React Native 0.21.0 版本的绝大部分特性。没有讨论的部分在书末有提及。

首先需要说明的是，本书讨论的知识以跨平台（Android 平台与 iOS 平台）开发为主，书中各章节的绝大部分知识点都是跨平台实现的。只有极小部分是分平台实现的，这一小部分在讨论前都会说明该部分知识适用于哪个平台。

本书的结构是按一个有基本的编程基础知识，刚学习 JavaScript 基本语法的读者最佳学习路径来编写的。通过一个个精简的例程，阐述清楚一个个基本思想。例程尽可能地精简，并且所有例程都是笔者花了大量时间为初学者构思而成的。

本书体例说明

代码与代码说明

本书例程中有大量的代码说明，通过注释的方式与代码同时展示出来。例如：

```
var IncomingCall = React.createClass({  
  watcher: null,           //用来记录监视器  
  startFromLeft:true,      //用来判断用户最先按下的是最左侧还是最右侧  
  moveNeedhandle:false,    //用来判断监测到的移动事件是否需要处理  
});
```

注释以黑体字显示以提醒读者注意。读者在自己的开发环境中输入例程时不需要输入注释。

React Native 代码中的 JSX 部分代码不能使用这种注释方式，本书为了统一注释风格还是使用了这种注释方式。读者明白了这一点后，在自己输入代码实践时，将代码中的注释自行去掉。

注意和提示

注意和提示，是需要提醒读者特别注意的内容，在本书中使用带背景色的字体显示。

致谢

感谢我的父亲阙光金老师与我的母亲袁雪英老师从小到大给我的无私的爱。很抱歉无论我如何努力，也无法回报二老深恩的万分之一。

感谢我的姐姐阙喜戎与姐夫王纯，没有你们的鼎力支持，就没有今天的我。

感谢王汝馨伯父与曾钰伯母，谢谢你们对我的关怀与照顾。

感谢廖建新教授、饶牧老师在我学习工作期间对我的关怀与指导。感谢在我七年北京邮电大学学习期间为我授业解惑的所有老师，谢谢你们！

感谢 React Native 开发团队，感谢所有参与 React Native 开发的贡献者。无数移动开发者因为你们无私的奉献而受益。

感谢电子工业出版社郭立总经理、孙学瑛编辑等审校此书的辛勤工作，以及为此书能快速出版而付出的巨大努力。你们辛苦了！

感谢在工作和生活中帮助过我的所有人，感谢你们，正是因为有了你们，才有了本书的面世。

关于勘误

虽然花了很多时间和精力去核对书中的文字、代码和图片，但因为时间仓促和水平有限，书中仍难免会有一些错误和纰漏，如果大家发现什么问题，请反馈给我，相关信息可在下载本书代码的 GitHub 页面反馈。

目 录

第 1 章 React Native.....	1
1.1 React Native 开发特点.....	2
1.1.1 一次学习，随处编写.....	2
1.1.2 混合开发.....	2
1.1.3 高效的 UI 开发.....	3
1.1.4 高效的 UI 调试.....	4
1.1.5 学习门槛低、开发难度低.....	4
1.1.6 开发软硬件要求低.....	5
1.1.7 使用 React Native 开发的代价.....	5
1.1.8 为什么 React Native 尚未流行.....	7
1.2 React Native 开发环境搭建.....	7
1.2.1 开发环境搭建起点.....	7
1.2.2 Windows 操作系统下 React Native 开发环境搭建.....	8
1.2.3 苹果操作系统下 React Native 开发环境搭建.....	9
1.2.4 查看与删除使用 npm 命令安装的软件.....	11
1.3 代码编辑环境搭建.....	11
1.3.1 Sublime Text 3.....	11
1.3.2 开发用插件.....	11
1.3.3 Sublime 界面风格选择.....	13
1.3.4 键盘使用习惯.....	13
1.4 React Native Dev tool 安装.....	14
第 2 章 状态机思维与状态机变量.....	16
2.1 初始化项目.....	16
2.2 运行项目.....	17
2.2.1 使用 Android 手机进行调测.....	18
2.2.2 使用 iPhone 手机或模拟器进行调测.....	21

2.2.3	修改 JSX 代码	22
2.2.4	ES 6 语法与 ES 5 语法	24
2.2.5	启动调试工具	25
2.3	构建注册页面	28
2.4	React Native 代码执行逻辑	32
2.5	UI 框架工作基本机制	33
2.5.1	状态机思维	33
2.5.2	“冒充常量”的状态机变量	35
2.5.3	“无处安放”的状态机变量	36
2.5.4	“努力瘦身”的状态机变量	36
2.6	React Native 组件间通信	37
2.7	深入理解 UI 重新渲染的过程	37
2.7.1	合并状态机变量	37
2.7.2	判断是否渲染	40
2.7.3	替换状态机变量	40
2.7.4	强制启动渲染	41
2.7.5	渲染过程	41
2.7.6	合并状态机变量的最简语法	41
2.8	React Native 组件的成员变量	42
2.9	React Native 组件的静态变量、静态函数	43
第 3 章	页面导航、弹出框及深入理解属性	44
3.1	分离注册组件、组件平台自适应	44
3.1.1	分离注册组件	44
3.1.2	组件平台自适应	44
3.1.3	平台检测	45
3.2	导航组件、挂接注册组件	45
3.3	挂接注册等待组件	47
3.4	Navigator 组件工作机制	49
3.4.1	push 与 pop	50
3.4.2	replace 函数	50
3.5	自定义组件	51
3.5.1	“弹出一切框”的实现	51
3.5.2	React Native 中颜色类型的值	53
3.5.3	挂接自定义组件	54
3.6	BackAndroid API 的 bug 与解决办法	56
3.7	属性确认	58
3.8	指定属性默认值	60

3.9	Alert 应用程序编程接口	60
3.9.1	弹出确认框	60
3.9.2	弹出选择框	61
3.10	带导航栏的页面导航	62
第 4 章	混合开发基础篇	63
4.1	iOS 平台混合开发	63
4.1.1	与 iOS 侧原生代码消息互通	64
4.1.2	React Native 代码到 iOS 原生代码的消息	65
4.1.3	iOS 原生代码到 React Native 代码的消息	68
4.1.4	与 iOS OC 原生代码界面的切换	69
4.1.5	应用初始界面设定	69
4.1.6	iOS 混合开发中传递的参数类型	70
4.1.7	混合开发中的多线程使用	70
4.1.8	原生代码实现 Promise 机制	71
4.1.9	跨语言常量	72
4.2	Android 平台混合开发	73
4.2.1	与 Android 原生代码消息互通	74
4.2.2	React Native 代码到 Android 原生代码的消息	75
4.2.3	与 Android 原生代码界面的切换	78
4.2.4	Android 原生代码到 React Native 代码的消息	82
4.2.5	应用初始界面设定	86
4.2.6	传递的参数类型	86
4.2.7	回调函数与 Promise 机制	86
4.2.8	监听 ActivityResult 与 Android 生命周期事件	88
4.2.9	混合开发中的多线程机制	89
4.2.10	跨语言常量	89
第 5 章	flexbox 布局、View、Image 与可触摸组件	90
5.1	flexbox 布局	90
5.1.1	位置及宽、高相关样式键	91
5.1.2	决定子组件排列规则的键	92
5.1.3	决定组件显示规则的键	94
5.1.4	边框、空隙与填充	95
5.1.5	组件多样式声明与动态样式声明	96
5.2	View 组件	97
5.2.1	View 组件的颜色与边框	97
5.2.2	View 组件的阴影与其他视觉效果	99

5.2.3	View 组件的变形	101
5.2.4	View 组件的回调函数	104
5.2.5	View 组件的其他属性	106
5.2.6	设备放置状态、根 View 与 onLayout 回调函数	106
5.2.7	pointerEvents 属性	109
5.3	Image 组件	111
5.3.1	加载网络图片	111
5.3.2	加载静态图片资源	112
5.3.3	加载资源文件中的图片	112
5.3.4	动态加载手机中的图片资源	112
5.3.5	Image 组件的样式	113
5.3.6	Image 组件显示特性	114
5.3.7	Image 组件的其他属性	117
5.4	可触摸组件	117
5.4.1	可触摸组件类型	118
5.4.2	TouchableOpacity 组件	118
5.4.3	TouchableHighlight 组件	118
5.4.4	其他属性	120
5.5	加深理解三大组件	120
5.5.1	使用导航栏的导航框架	121
5.5.2	等比放大无丢失显示图片	125
5.5.3	宽、高动态变化的组件呈现	128
第 6 章	Text、TextInput 等相关知识	129
6.1	Text 组件	129
6.1.1	样式键设置	129
6.1.2	其他属性	131
6.1.3	Text 组件的嵌套	131
6.1.4	文本显示的阴影效果	132
6.1.5	Text 居中显示	133
6.1.6	在字符串中插入图像	135
6.2	Text 组件在两个平台上的不同表现	136
6.2.1	只指定 fontSize, 不指定 height	137
6.2.2	只指定 height, 不指定 fontSize	137
6.2.3	fontSize 等于 height	137
6.2.4	height 大于 fontSize	138
6.2.5	边框在两个平台上的不同表现	138
6.3	TextInput 组件	140

6.3.1	TextInput 组件样式键	140
6.3.2	TextInput 组件的属性	140
6.3.3	TextInput 组件 iOS 平台专有属性	141
6.3.4	TextInput 组件 Android 平台专有属性	142
6.3.5	TextInput 组件的成员函数	142
6.4	TextInput 组件在两个平台上的不同表现	143
6.4.1	Android 平台的输入下画线	143
6.4.2	父组件的 alignItems 键失效	144
6.4.3	只指定 fontSize、不指定 height	145
6.4.4	height 等于 fontSize	145
6.4.5	height 大于 fontSize	146
6.4.6	边框在两个平台上的不同表现	146
6.5	TextInput 组件的生命周期	147
6.5.1	获得焦点	147
6.5.2	用户输入	147
6.5.3	用户按下提交键	147
6.5.4	失去焦点	148
6.6	软键盘与键盘事件	148
6.7	组件的引用	151
6.7.1	定义组件引用	151
6.7.2	得到系统定义的组件引用	151
6.7.3	调用组件的公开成员函数	152
6.7.4	重新设定组件的属性	152
6.7.5	获得组件的位置	154
6.8	跨平台状态栏组件	155
6.8.1	StatusBar 组件属性	155
6.8.2	StatusBar 组件使用示例	156
6.8.3	手机状态栏在开发中的处理	157
6.8.4	StatusBarIOS API	158
6.9	高度自增长的扩展 TextInput 组件	159
6.10	访问操作系统剪贴板	160
第 7 章	组件生命周期、数据存储及 React Native 应用实现步骤	163
7.1	组件生命周期	163
7.1.1	getInitialState	163
7.1.2	getDefaultProps	163
7.1.3	componentWillMount	164
7.1.4	componentDidMount	164

7.1.5	componentWillReceiveProps	164
7.1.6	shouldComponentUpdate	165
7.1.7	componentWillUpdate.....	165
7.1.8	componentDidUpdate.....	165
7.1.9	componentWillUnmount	166
7.2	读取 JSON 文件	166
7.3	数据持久化操作.....	167
7.3.1	flow 语法检查器.....	167
7.3.2	写入数据、错误捕捉	168
7.3.3	读取数据	170
7.3.4	AsyncStorage API 存储数据的无序性	173
7.3.5	删除数据	173
7.3.6	修改数据	175
7.3.7	JSON 对象存储	175
7.3.8	读取 JSON 对象	176
7.4	数据表操作.....	176
7.5	React Native 应用实现步骤、日记例程（上）	176
7.5.1	应用原型	177
7.5.2	基础组件结构设计	178
7.5.3	使用 React Native 组件搭建静态界面	180
7.5.4	React Native 组件分层	188
7.5.5	实现各组件业务逻辑	189
7.5.6	日记例程（上）总结	190
第 8 章	ScrollView 和 ListView.....	200
8.1	ScrollView 组件	200
8.1.1	ScrollView 组件属性.....	200
8.1.2	ScrollView 组件 iOS 平台专有属性.....	201
8.1.3	ScrollView 组件 Android 平台专有属性.....	202
8.1.4	ScrollView 组件的公开成员函数.....	203
8.1.5	RefreshControl 组件	203
8.1.6	ScrollView 组件基本用法	204
8.2	ListView 组件.....	206
8.2.1	ListView 组件的回调函数	207
8.2.2	ListView 组件的其他属性	208
8.2.3	ListView 组件的成员函数	208
8.3	简单的列表.....	208
8.3.1	准备列表的数据源	209

8.3.2	声明状态机变量	209
8.3.3	将数据源中的数据拷贝到 DataSource 中	210
8.3.4	定义如何渲染列表中的每一行	210
8.3.5	实现简单的列表	211
8.3.6	列表栏的高级处理	217
8.4	带分段标志的列表	217
8.4.1	准备数据源	218
8.4.2	声明状态机变量	218
8.4.3	将数据源中的数据拷贝到 DataSource 中	219
8.4.4	定义如何渲染每个分栏	219
8.4.5	定义如何渲染首、尾栏	220
8.4.6	列表间隔渲染	220
8.4.7	实现带分段标志的列表	220
8.4.8	总结	221
8.5	日记例程（下）总结	221
第 9 章	等待提示条、进度条和 Switch	222
9.1	ProgressBarAndroid 组件	222
9.1.1	ProgressBarAndroid 组件样式设置	222
9.1.2	ProgressBarAndroid 其他属性	222
9.1.3	Android 平台等待提示条	222
9.1.4	React Native 框架中定时器的使用	224
9.1.5	Android 平台进度条	225
9.2	iOS 进度条组件	226
9.2.1	ProgressViewIOS 组件样式设置	226
9.2.2	ProgressViewIOS 其他属性	226
9.2.3	iOS 平台进度条	226
9.3	iOS 平台等待提示条	227
9.3.1	ActivityIndicatorIOS 组件样式设置	227
9.3.2	ActivityIndicatorIOS 其他属性	227
9.3.3	iOS 平台等待提示条例程	227
9.4	Switch 组件	229
9.4.1	Switch 组件样式设置	229
9.4.2	Switch 其他属性	229
9.4.3	Switch 组件的使用	229
第 10 章	导航组件	231
10.1	导航组件的属性	231

10.1.1 回调函数	231
10.1.2 其他属性	232
10.2 导航器	232
10.3 NavBar	233
第 11 章 手势识别	240
11.1 PanResponder API	240
11.2 监视器	240
11.2.1 指定监视区域	241
11.2.2 定义监视器相关变量	241
11.2.3 准备监视器的事件处理函数	241
11.2.4 建立监视器	242
11.2.5 将监视器与监视区域挂接	242
11.3 监视事件的生命周期	242
11.3.1 单次点击事件的生命周期	243
11.3.2 单次点击事件处理	245
11.3.3 移动手势事件的生命周期	245
11.3.4 监视器异常事件	247
11.4 手势识别处理例程	247
11.4.1 单点手势——点击、拖动选择百分比参数	247
11.4.2 单点手势——带导槽的滑动来电接听或拒接界面	249
11.4.3 单点手势——滑动解锁屏幕界面	252
11.4.4 单点手势——单点任意方向拉动选择界面	254
11.4.5 两点手势	257
第 12 章 网络	258
12.1 获取网络状态	258
12.1.1 得到当前网络状态	258
12.1.2 监听网络状态改变事件	259
12.1.3 简单判断是否有网络连接	260
12.1.4 判断当前连接是否收费	260
12.2 通过 HTTP、HTTPS 与网络侧交换数据	260
12.2.1 发送请求	260
12.2.2 接收响应	263
12.3 在 React Native 开发中使用 AJAX 技术	264
第 13 章 网页浏览器、音视频媒体播放	266
13.1 WebView 组件样式设置	266

13.2	WebView 组件其他属性.....	266
13.2.1	非回调函数属性.....	266
13.2.2	回调函数属性.....	267
13.2.3	平台独有属性.....	267
13.2.4	WebView 组件成员函数.....	268
13.3	网页浏览器使用例程.....	268
13.3.1	浏览网页例程.....	268
13.3.2	加载本地网页例程.....	271
13.4	音视频媒体播放.....	273
第 14 章	图片的遍历、存取与显示	274
14.1	React Native 开发中 iOS 平台链接库的使用	274
14.2	获取手机中所有的图片信息.....	276
14.3	图片信息详解.....	278
14.3.1	Android 平台图片信息.....	278
14.3.2	iOS 平台图片信息.....	278
14.4	显示从 CameraRoll API 得到的图片	279
14.5	为用户提供图片选择界面.....	280
14.6	图片的保存与读取显示.....	282
14.6.1	保存图片数据.....	282
14.6.2	读取并显示图片.....	283
第 15 章	选择器、位置相关和应用状态	284
15.1	日期、时间选择器.....	284
15.1.1	DatePickerAndroid API	284
15.1.2	TimePickerAndroid API.....	286
15.1.3	DatePickerIOS 组件.....	287
15.2	Picker 组件	289
15.2.1	Picker 组件的样式设置.....	289
15.2.2	Picker 组件的属性.....	289
15.2.3	Picker.Item 组件属性	290
15.2.4	Picker 组件例程.....	290
15.3	PickerIOS.....	294
15.4	MapView 组件.....	295
15.4.1	MapView 组件样式设置	296
15.4.2	MapView 组件特有的跨平台属性	296
15.4.3	MapView 组件例程.....	297
15.5	AppState API	299

15.5.1	AppState API 用途与用法	299
15.5.2	AppState API 例程	299
15.6	获取地理位置	300
15.7	VibrationIOS API	302
第 16 章	使用 ES 6 语法开发	303
16.1	React Native 组件导入	303
16.2	属性声明	304
16.3	成员变量声明	304
16.4	状态机变量声明	305
16.5	回调函数绑定	306
16.6	类的静态成员变量与静态成员函数	307
第 17 章	混合开发高级篇	309
17.1	使用 Objective-C 语言创建私有的 React Native 组件	309
17.1.1	增加 FLAnimatedImage 链接库	309
17.1.2	创建视图管理类	311
17.1.3	封装开源代码中的视图类	312
17.1.4	在 React Native 侧调用私有组件	314
17.1.5	例程运行效果	315
17.2	使用 Swift 语言创建私有的 React Native 组件	316
17.2.1	整合开源项目	316
17.2.2	建立组件管理者和桥接头文件	319
17.2.3	Objective-C 与 React Native 接口部分	321
17.2.4	使用 Swift 语言实现组件控制	322
17.2.5	在 React Native 侧调用私有组件	324
17.2.6	例程运行效果	325
17.3	使用 Android SDK 创建私有的 React Native 组件	325
17.3.1	准备原生代码 UI 组件	326
17.3.2	实现原生 UI 管理类	327
17.3.3	创建原生 UI 实例	328
17.3.4	实现对属性的支持	328
17.3.5	建立原生 UI 包	328
17.3.6	注册原生 UI 管理类	329
17.3.7	对应的 React Native 侧实现	329
17.3.8	运行俯视视图例程	331

第 18 章 项目配置、生成发布版本安装包及其他.....	332
18.1 iOS 平台项目配置	332
18.2 iOS 平台应用发布	336
18.3 Android 平台项目配置	336
18.4 Android 平台应用生成发布版本安装包.....	338
18.4.1 生成发布密钥.....	338
18.4.2 修改 gradle 配置文件.....	338
18.4.3 生成发布版本安装包.....	339
18.5 其他组件与 API	339
18.5.1 动画相关.....	339
18.5.2 其他未讨论的组件与 API.....	340
附录 A ECMAScript 2015 语法参考.....	341

第 1 章

React Native

2015 年 3 月 26 日，Facebook 公司对外正式发布了 React Native——使用 React 框架跨平台开发原生移动应用的开源技术框架。开发者可以使用 React Native 高效地开发运行于 Android 与 iOS 操作系统的应用程序。它的设计理念是：使用 React Native 开发，既拥有 Native 的良好人机交互体验，又保留了 React 框架的开发效率。

提示：在这句话中，Native 是指使用原生开发环境开发的应用程序。Native 的良好人机交互体验是指当用户用手指在屏幕上操作时，被触摸的 UI 组件会发生视觉上的变化（比如某项变暗或者高亮、长按时被按的部分出现动画效果、拖动列表到头时的弹回效果、手机的震动效果等）。

相对于类 Web 界面的应用程序，Native 的良好人机交互体验应该具有以下属性：

- 反馈/高亮——显示给用户是什么正在处理他们的触摸，以及当他们释放手势时会发生什么；
- 撤销的能力——在做一个动作的过程中，用户应该能够在触摸过程中通过移动手指中止该动作。

Native 的良好人机交互体验让用户使用一个应用程序时更舒适，因为它允许用户在体验和交互时不用担心犯错误。

而 React 是 Facebook 从 2012 年以来慢慢发展起来的一套开发框架。在这套框架上诞生了 React.js 用来进行网页开发，以及 React Native 用来进行手机 APP 开发（它们三者的关系在本书前言的“如何阅读本书”中有详细描述，如果你阅读时跳过了前言，强烈建议你现在就去阅读）。使用 React.js 开发网页的效率比普通的 HTML 网页编写要高出非常多。它现在也是一项非常热门的网页开发技术。React Native 的开发效率，读者很快就能体验到。

React 框架不追求所谓的“一次编写，随处运行（Write once, run anywhere.）”。React 认为不同的平台应该有不同的外观、感觉、功能等。开发者仍然需要为不同的平台去做一些额外的工作。React 把不同平台的能力分为跨平台通用能力与平台特色能力，这样应用程序的代码也分成了跨平台部分与平台特色部分。React 把这个方案叫作“一次学习，随处编写（Learn once, write anywhere.）”。

1.1 React Native 开发特点

在 React Native 发布的短短 5 个月里，就有 60 多个使用 React Native 技术开发的 APP 在苹果软件商店上线。开发者在尝试后对 React Native 赞不绝口。它究竟有哪些优点呢？

1.1.1 一次学习，随处编写

在 iOS 与 Android 这两个操作系统上实现统一的开发框架，一份代码支持两个操作系统在历史中已经证明是非常困难的。React Native 提出了“Learn once, write anywhere.”。使用 React Native 可以为这两个操作系统开发应用程序，但不同平台上的代码根据平台会有一些微小区别，但开发思路是相同的。只需要根据平台进行一些代码调整，有经验的开发人员进行这种调整的速度非常快。React Native 开发小组没有狂妄地喊出“Write once, run anywhere.”，但在 UI 开发上，React Native 差不多已经做到了这一点。

1.1.2 混合开发

混合开发是 React Native 的另一个重要特性。React Native 允许开发者在 React Native 擅长的领域使用 React Native 开发，而在 React Native 不能实现的领域或者已经有原生代码实现好的领域直接使用原生代码。React Native 代码开发的模块与原生代码开发的模块可以双向通信、无缝衔接。这使混合开发变成了件很轻易的事。

很多读者不是很熟悉混合开发这个概念，在这里详细说明一下。混合开发可以分为 3 种情况：

1. UI 界面由 React Native 开发，但 UI 事件处理由原生代码来执行

举个简单的例子。在 React Native 开发的界面上有让用户输入用户名与密码的 UI 控件，还有一个登录按钮。用户点击登录按钮后，React Native 组件将用户输入的用户名与密码传给原生代码编写的登录模块（在 Android 上，用 Java 语言开发；在 iOS 上，用 Objective-C 或者 Swift 语言开发），让原生代码执行登录操作。原生代码通过互联网向网络侧的服务器发送登录消息，并等待服务器回应。在服务器回应后，原生代码再将收到的回应中的登录成功与否，以及其他一些需要 UI 展示的数据传递给 React Native 组件，React Native 组件接收原生代码传来的数据，解析这些数据并执行 UI 界面更新，向用户展示相关信息。

2. 将原来使用原生代码实现的 UI 小部件包装成 React Native 的自定义组件

Widget 在移动应用程序开发中被广泛使用，它们有官方发布的，也有第三方开源的，还有开发者自行开发的。React Native 支持将这些 Widget 包装成 React Native 的自定义组件，然后就可以在 React Native 代码中方便地使用了。

3. 应用界面在 React Native 开发的界面与原生代码开发的界面间切换

在某些情况下，我们希望使用原生代码开发的界面，比如某个界面，在原来的版本中已经开发好了，或者希望在已经用原生代码开发好的项目中加入一些用 React Native 开发的新界面。混合开发可以做到让应用界面流畅自如地在这两种界面间切换，用户对此不会有任何感知。

我们可不可以用原生代码来开发 UI 界面，让 React Native 模块处理 UI 事件呢？理论上是可以的，但很少有人这么干。因为 React Native 的强项就是 UI 开发，在混合开发中，能用 React Native 开发的界面，优先用 React Native 开发。

提到混合开发，不得不提到另一个分支：在手机上使用 WebView 来呈现部分 UI 界面。使用 WebView 开发比较灵活，能沿用全部 Web 开发的习惯，比如 React.js 的各种好处和 Web 的快速迭代流程。但因为所有的渲染都由 Web 相关技术来完成，使用 WebView 无法得到真正原生的用户体验，并且 WebView 无法做到与原生代码双向通信、无缝衔接。React Native 不排斥 WebView 开发，并且为 WebView 提供了相应的组件，可以在 React Native 中实现部分界面通过 WebView 呈现。

1.1.3 高效的 UI 开发

对于移动应用开发来说，在单个平台上，UI 部分开发工作量占移动应用开发总工作量的比重至少是 50%。对于追求界面美观、使用方便、容易上手的移动应用来说，这个比例会提高到 70% 左右。再考虑到很多应用都需要兼顾 Android 与 iOS 两个平台，UI 开发的工作量又被放大了 1 倍。

在这个时候，使用 React Native 开发的优势就显露无遗了。使用 React Native 开发移动应用的 UI 界面比使用原生语言快捷高效，再考虑到至少 90% 的移动应用界面都可以使用 React Native 开发，一份代码适配 Android 与 iOS 两个平台，这相当于减掉了一个开发平台上至少 50% 的工作量。开发者找不到任何理由不使用 React Native 开发移动应用。

虽然 React Native 可以实现很多 UI 之外的功能，但开发 UI 部分绝对是 React Native 的强项。这体现在 4 个方面上。

1. 独特的 UI 实现框架

React 重新思考了 UI 开发过程，并且提出了一套全新、高效的框架。复杂 UI 界面开发难点的本质问题是：如何将来自于网络侧与用户侧的动态数据高效、实时、正确地呈现在复杂的用户界面上。Facebook 的 React 框架是这个问题的一个优秀解决方案。React Native 官网描述它的出发点为：用于开发数据不断变化的大型应用程序（Building large applications with data that changes over time）。相比传统型的 UI 开发，React 开辟了一个相当另类的途径，实现了 UI 界面的高效率、高性能开发。使用 React 开发，开发者永远只需要关心数据。当数据改变时，开发者只需要告诉 React 数据变了，由 React 来实现 UI 界面的改变。本书一进入正题在第 2、3 章重点讨论的状态机变量与属性就是这个框架的基石之一。

2. 组件化开发

React 推荐以组件的方式去重新思考 UI 构成，将 UI 上每一个功能相对独立的模块定义成组件，然后将小的组件通过组合或者嵌套的方式构成大的组件，最终完成整体 UI 的构建。

React 强调将应用划分成多个互不相关的组件，每个组件作为一个独立的视图。这使得开发者更容易进行软件迭代升级，因为不用在改动某一小部分时把整个系统都梳理一遍。最重要的是，

React 包装了复杂而易变的数据到对象的实现，改为提供一个声明式的结构，使得整个程序模型变得抽象而简单。使用 React 来构建网页或者手机 UI 时，代码变得容易预测。这种可预测性使得开发者在快速地迭代产品时可以更多地信任已有的代码，最终应用程序也变得更为可靠。更进一步的，React 框架不仅仅使扩大应用规模变得更容易，也使团队规模更容易进行调整。

这样开发出来的代码结构清晰、共用性高、可移植性高。上一个项目的某些组件，可以很方便地拿来在下一个项目中使用；在网上看到的好的开源组件，可以下载后很方便地集成在我们的项目中使用。

3. 跨平台移植代码迅速

使用 React Native 进行 UI 开发时，开发者通常先在一个操作系统（iOS 或 Android）上开发，然后用这套代码去另一个操作系统中进行修正。通常需要修正的代码只有总代码的 5% 甚至更低。有经验的开发者会很清楚两个系统有哪些不同，代码中的哪些部分需要修改，快速地实现 React Native 代码对两个操作系统的适配。

4. 自动匹配不同屏幕大小的手机

在手机屏幕上布局、显示一个 UI 组件十分困难，开发者往往不得不自己去计算视图的大小和位置，然后让 UI 组件去适应视图的大小。

使用 React Native 开发，开发者无须为不同的屏幕分辨率准备不同的图片资源或者布局文件，甚至可以不考虑屏幕大小的问题。通过灵活的布局方式，React Native 可以做到在不同的手机屏幕上高效、清晰的 UI 呈现。

1.1.4 高效的 UI 调试

在原生开发过程中，开发者的每一次改动（即使改动的元素非常小，如一个单词，或者一个位置）都需要经历重新编译和构建，然后把安装包上传到手机的过程，这使得开发者在做很多工作时变得非常缓慢，尤其是当一个大工程的编译特别慢时。

使用 React Native 开发，修改了代码后立刻可以在手机上看到效果，没有重新编译启动程序所需要的时间。并且可以打开一个 Chrome 窗口，所有的代码都移到 Chrome 里面运行，断点调试、单步调试、调用栈追踪这些常用的调试方法都可以进行操作。

1.1.5 学习门槛低、开发难度低

看到技术大牛们的巨著都是厚厚的一本，笔者也东施效颦，非常努力地尝试着把这本书写得厚些，但本书还是这么薄！从这点读者就应当知道，使用 React Native 开发的难度实在很低。下面还是总结一下为什么使用 React Native 开发的难度低。

1. 开发语言简单

React Native 使用 ECMAScript 2015（虽然它穿了件很华丽的“马甲”，但我们还是一眼就认

出来它就是 JavaScript 语言(也简称为 ES 6),以及自创的 JSX 语言(通俗些说,就是在 JavaScript 中加入一些标签化的 XML)来进行开发。JavaScript 是一门使用很广泛的语言。相对于目前手机开发人员不足的情况,JavaScript 开发人员却很多。经过简单的学习,没有移动应用程序开发基础的 JavaScript 开发人员就能使用 React Native 进行移动应用程序的 UI 与部分业务逻辑的开发了。

对于没有 JavaScript 知识背景的手机 APP 开发人员,只需要用几天时间熟悉 JavaScript 的基本语法后就可以使用 React Native 进行开发。

JavaScript 是一门很特殊的语言,它很简单但也可以非常复杂。React Native 从一开始就注意到了 JavaScript 的不足之处,要求在 React Native 开发中使用 JavaScript 的“严格模式”,并且采用更先进的 ECMAScript 2015,能够做到取其精华,去其糟粕。

而 React Native 使用的 JSX 语言也是非常简单的,简单到本书不会有专门的章节去讲解它的语法。读者只要一个一个例程从浅入深地学下去,不知不觉间,就已经掌握 JSX 语言了。

2. 语法接近自然语言

React Native 开发中的函数名、变量名都采用类似于自然语言的命名法,便于记忆。这种代码,语句的含义基本上可以直接推断与理解。因此学习简单,容易上手。示例详见代码 1-1。

代码 1-1:

```
.....
componentDidMount: function() {
.....
componentWillUnmount: function() {
.....
  propTypes: {
    userConfirmed: React.PropTypes.func.isRequired,
    promptToUser: React.PropTypes.string.isRequired,
  },
.....
```

3. 积木式 UI 开发

使用 React Native 开发 UI 时,是一种类似于搭积木的方式。不论是设计还是实现,通过 React Native 框架都能做到逻辑结构清晰、开发难度低、可读性高、后期修改维护方便。

1.1.6 开发软硬件要求低

使用 React Native 开发对软硬件要求低,这意味着开发者不需要再掏钱购买电脑硬件与相关软件。React Native 开发用软件都是可免费下载、安装使用的正版软件,部分是开源软件。对硬件要求也不高。

我们将在 1.2 节中详细讨论开发的硬件要求与开发环境搭建。

1.1.7 使用 React Native 开发的代价

为了得到 React Native 开发的优点,使用 React Native 开发的 APP 也需要付出一些代价。代价

体现在三个方面上。

1. 内存消耗略大

使用 React Native 开发的程序运行所需的内存比原生代码开发的程序略多。会多多少，没有人认真分析过，笔者也不打算认真分析。

为什么没人愿意分析内存消耗情况呢？因为现在是手机硬件配置已经很强，并且还会越来越强的时代。在笔者写本书时，市场上 700 元级的入门 Android，手机内存配置都达到了 2GB，1500 元级的中低端手机内存配置达到了 4GB。手机用户基本上感觉不到应用程序多占了几十兆内存。某些知名购物、支付移动应用 APP 在运行时使用的内存已经达到了 500MB 左右，但根本就没有听到用户对此有任何抱怨。用吧，用了 500MB 内存，也没有给用户带来任何损失；不用那么多内存，也不能给用户省下一分钱。

开发者开发调试时，React Native 项目通常运行在“开发模式”下，这时最简单的 Hello World 程序会比原生代码的 Hello World 程序多用 20MB 内存，这是正常的。当使用发布模式编译项目后，React Native 项目占用的内存会比开发模式小很多，最简单的 Hello World 程序会与原生代码的 Hello World 程序消耗的内存相差不大。

2. 运行速度

同样一个应用程序，让一个原生语言开发高手用该手机原生语言开发出来的版本比使用 React Native 框架开发出来的版本运行速度要略快。但如果只是普通的开发者，因为 React Native 框架的先进性，使用 React Native 开发出来的版本的运行速度与原生语言开发出来的版本差别不大，甚至会更快一些、Bug 更少一些。

使用 React Native 开发的代码的运行速度比原生代码略慢。速度慢的缺点可以通过两方面来弥补。一是普通的功能（如 UI 展示、HTTP 请求等），React Native 实现的速度比原生代码慢，但用户感觉不出来，因此不需要加快。比如显示一个页面，原生代码用 10ms 完成，React Native 代码用了 11ms，这对用户来说没有区别。再比如从网络获取数据，这个操作耗时的大头是网络传输时延，用什么语言实现对加快获取都没有帮助。二是核心的功能，通过原生代码来开发，也就是混合开发移动应用程序。

需要特别指出的是，开发者开发调试时，React Native 项目通常运行在“开发模式”下，因为有很多特殊的任务需要执行（例如：验证属性类型，产生各种调试信息与警告信息，显示这些信息），代码的运行速度要比“发布模式”下的代码运行速度慢。

3. 安装包比原生代码安装包大

使用 React Native 开发的程序体积比原生代码大。不论是安装包的大小，还是安装后所需的空间都比原生代码编写的程序要大。这勉强算是一个使用 React Native 开发的代价吧。现在手机存储容量通常 64GB，还可以通过存储卡扩充，因此这个代价微不足道。

1.1.8 为什么 React Native 尚未流行

看到这里，估计很多读者都会想既然 React Native 这么好，为什么还没有开始流行呢？你不会是在“坑儿”我们吧！

React Native 尚未流行的原因有两个。第一个原因是 React Native 于 2015 年 3 月发布时就说目标是支持两大主流手机平台，但直到 2015 年 11 月 6 日放出的 0.14.0 版本才正式支持 Android 平台。这是 React Native 的一个重大里程碑。从这一天起，它才是真正的跨平台开发框架。它的出现时间还不长。

第二个原因是主要原因。React Native 对 Android 的支持正越来越好，但它还是需要一个提高的过程。在初期，React Native 对手机配置较高、Android 操作系统版本高于 5.0 的 Android 手机的支持比较好。而对 Android 操作系统低于 5.0 的 Android 手机的支持还有待完善。也就是说，在 2016 年年初，使用 React Native 框架开发 Android 移动应用程序在老手机上运行还是会遇到问题。

但移动应用开发总是向前发展的，当读者看到这本书时，应当是 2016 年年中，读完这本书，开始用 React Native 框架开发应用的时间应当是 2016 年第三季度。到了那个时候，一方面是 React Native 对 Android 低版本手机支持会更好；另一方面也得益于 Android 手机的廉价，它的更新速度快，那时低版本 Android 手机已经不会再是市场保有量的主流。所以到了 2016 年年底，使用 React Native 框架进行跨平台移动应用开发必然会流行起来。不仅是流行起来，而且是成为商业移动应用开发的主流方式。

1.2 React Native 开发环境搭建

本节将介绍 React Native 开发环境的搭建。React Native 开发环境可以搭建在 Windows 平台或者苹果平台下。早期 React Native 对苹果平台的支撑比较好，但到 0.21.0 版本，在 Windows 平台下 React Native 开发环境也很成熟了。

1.2.1 开发环境搭建起点

React Native 开发环境可以在 Windows 或者苹果电脑上搭建。不管是使用苹果电脑还是 Windows 电脑开发，电脑的 CPU 最好不要低于 Intel i5 这个档次，内存不要低于 8GB（现在 8GB 内存最贵也就 300 元钱，如果内存不够建议添加一条）。硬盘剩余空间现在谁也不缺，这里就不说需要多少了

在 Windows 操作系统下搭建 React Native 开发环境要求已经在电脑上安装好 Java、Android SDK，还要求电脑上安装有一套 C++ 编译器。如果没有，笔者推荐安装微软的 Visual Studio Community 2015（可以从微软网站免费下载安装，只是安装光盘镜像文件比较大，近 6GB）。

Android 的开发环境 Android Studio（推荐）或者 Eclipse 在混合开发中也需要用到。

在苹果操作系统下搭建开发环境要求首先通过苹果的 APP Store 将苹果操作系统升级到最新版本，Xcode 也安装、升级到最新版本。Xcode 安装好后需要启动一次，同意苹果的开发者协议，

再自动补充安装一些 Xcode 开发组件。至此，iOS 开发环境已经搭建好。建议再搭建一个苹果电脑下的 Android 开发环境。对搭建步骤不熟悉的读者可以参考附录 A。

1.2.2 Windows 操作系统下 React Native 开发环境搭建

React Native 开发环境可以在 Windows 操作系统下搭建，但建议读者在 Mac 操作系统下搭建 React Native 开发环境。在 Windows 操作系统下搭建 React Native 开发环境的最大问题就是无法进行 iOS 平台的测试。如果读者是经济能力有限的在校大学生，只有 Windows 电脑，那么仍然可以在 Mac 操作系统下搭建开发环境（不建议使用“黑苹果”），相关方法请读者自行上网搜索。

React Native 各版本的安装步骤可能会随着新版本发布而有所改变，但是读者仍然可以参考本节讲述的安装步骤，特别是如何使用管理员身份运行命令行窗口，然后进行各项安装。

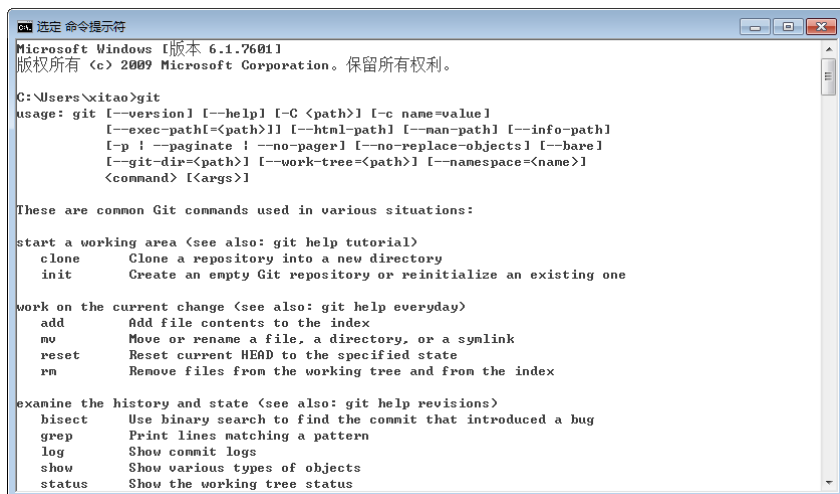
当需要安装时，请阅读官方文档上关于 React Native 最新版本的说明。它的网址是：<http://facebook.github.io/react-native/docs/getting-started.html>。中文网址是：<http://reactnative.cn/docs/getting-started.html#content>。

React Native 0.19.0 版本的安装步骤如下：

首先需要安装 Node.js，这是一个免费的软件，可以进入 www.nodejs.org 网站下载与安装。安装版本只要大于 4.0 就可以，但是推荐安装 5.0 以上的稳定版本。

然后进入 <http://www.git-scm.com/downloads> 页面，下载 git 软件并安装。

安装完成后，或者你的电脑上已经安装了 git，那么请验证 git 软件是否安装正确。在命令行窗口输入 git，然后按回车键。如果命令行窗口输出如图 1-1 所示，则表示安装正确。



```
选定 命令提示符
Microsoft Windows [版本 6.1.76011]
版权所有 (c) 2009 Microsoft Corporation。保留所有权利。

C:\Users\xitao>git
usage: git [--version] [--help] [-C <path>] [-c name=value]
           [--exec-path=<path>] [--html-path] [--man-path] [--info-path]
           [-p | --paginate | --no-pager] [--no-replace-objects] [--bare]
           [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
           <command> [<args>]

These are common Git commands used in various situations:

start a working area (see also: git help tutorial)
   clone      Clone a repository into a new directory
   init       Create an empty Git repository or reinitialize an existing one

work on the current change (see also: git help everyday)
   add        Add file contents to the index
   mv         Move or rename a file, a directory, or a symlink
   reset      Reset current HEAD to the specified state
   rm         Remove files from the working tree and from the index

examine the history and state (see also: git help revisions)
   bisect     Use binary search to find the commit that introduced a bug
   grep       Print lines matching a pattern
   log        Show commit logs
   show       Show various types of objects
   status     Show the working tree status
```

图 1-1 验证 git 安装是否正确

接下来需要以管理员身份启动命令行窗口。

打开 Windows 操作系统安装盘下的\windows\system32 文件夹，选择 cmd.exe 文件并单击鼠标

右键，选择以管理员身份运行。

提示：打开 system32 文件夹后，在键盘上快速连续按下 c,m,d 三个键，可以马上定位到 cmd.exe。

在命令行窗口输入 `npm install -g nrm`。安装 `nrm` 模块可以方便我们切换 `npm` 下载源。

在命令行窗口输入 `npm install -g npm@2` 来安装 `npm 2`。

如果网速足够快，并且访问国外网站没有什么限制，那么接下来就可以直接输入命令 `npm install -g react-native-cli` 来安装 `React Native`。

如果访问国外网站比较慢，那么先输入命令：

```
npm install -g cnpm --registry=https://registry.npm.taobao.org
```

这条命令创建了一个名为 `cnpm` 的安装点，这个安装点将从淘宝网提供的 `npm` 镜像站点安装 `React Native`。在后面介绍到初始化项目时，也是从淘宝网镜像下载的。

再输入：

```
cnpm install -g react-native-cli
```

1.2.3 苹果操作系统下 React Native 开发环境搭建

`React Native` 各版本的安装步骤可能会随着新版本发布而有所改变，但是读者仍然可以参考本节讲述的安装步骤，特别是前期软件包安装中需要输入的命令。

当需要安装时，请阅读官方文档上关于 `React Native` 最新版本的说明。它的网址是：<http://facebook.github.io/react-native/docs/getting-started.html>。中文网址是：<http://reactnative.cn/docs/getting-started.html#content>。

首先将苹果操作系统升级到最新版本，然后安装最新版本的 `Xcode`。

安装后，启动 `Xcode`，同意它的用户协议，`Xcode` 会有一个组件安装过程。

`Xcode` 组件安装完成后，关闭 `Xcode`。

进入命令行，首先需要安装 `Homebrew`，输入：

```
ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

中途需要按回车键，然后输入超级用户口令。

等待，直到窗口中出现 “Installation successful!”。

接下来需要安装 `Node.js`。进入 <https://nodejs.org>，安装最新稳定版本的 `node.js`。

安装结束时会出现提示：Make sure `/usr/local/bin` is in your path。对于苹果电脑，`/usr/local/bin` 已经被设置在 `PATH` 环境变量中了，不需要用户自己设置。

然后需要安装 `NVM`，在命令行输入：

React Native 跨平台移动应用开发

```
curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.29.0/install.sh | bash
```

或者

```
wget -qO- https://raw.githubusercontent.com/creationix/nvm/v0.29.0/install.sh | bash
```

安装完成后，在命令行窗口可能会显示：

```
Checking connectivity... done.
```

```
* (HEAD detached at v0.30.1)
```

```
master
```

```
=> Profile not found. Tried (as defined in $PROFILE), ~/.bashrc, ~/.bash_profile,
~/.zshrc, and ~/.profile.
```

```
=> Create one of them and run this script again
```

```
=> Create it (touch ) and run this script again
```

```
OR
```

```
=> Append the following lines to the correct file yourself:
```

```
export NVM_DIR="/Users/你的用户名/.nvm"
```

```
[ -s "$NVM_DIR/nvm.sh" ] && . "$NVM_DIR/nvm.sh" # This loads nvm
```

```
=> Close and reopen your terminal to start using nvm
```

如果出现了提示，则按提示修改用户目录下的`.bash_profile`。首先在命令行窗口用鼠标选中提示的倒数第三句与第二句话，单击右键选择拷贝。然后在命令行输入：

```
vi .bash_profile
```

此时会出现编辑窗口。在窗口中按 `i` 键，进入插入模式，单击鼠标右键选择“粘贴”（或者直接按“`Command+V`”快捷键）。粘贴完成后，按 `Esc` 键，再输入“`:wq`”，按回车键。回到命令行，完成修改。

接下来输入 `exit`，关闭命令行窗口。

再打开一个新的命令行窗口，输入 `echo $NVM_DIR` 并按回车键。如果命令行窗口中出现：`/Users/“你的用户名”/.nvm`，就说明修改正确完成，NVM 正确安装。

然后在命令行输入：

```
nvm install node && nvm alias default node
```

接下来需要安装 `flow` 软件包。但在安装前需要先在命令行输入：

```
sudo chown -R $USER /usr/local/lib
```

```
sudo chown -R $USER /usr/local
```

输入了这两行命令后，接下来的安装过程才不会出错。然后在命令行输入：

```
brew install watchman
```

```
brew install flow
```

最后，我们可以安装 `React Native` 了。在命令行输入：

```
sudo npm install -g react-native-cli
```

按回车键后，需要输入用户口令，完成安装。

1.2.4 查看与删除使用 npm 命令安装的软件

在 Windows 与 Mac 平台下,使用“`npm install -g ...`”安装的软件,可以使用“`npm list -g`”命令查看。开发者可以查看到哪些开发依赖包已经被全局安装,并且可以查看到被全局安装的开发依赖包的版本号。

使用“`npm install -g ...`”命令全局安装的开发依赖包,可以使用“`npm uninstall -g 依赖包名`”命令来删除。例如:如果开发者不小心输入了“`npm install -g react-native`”(命令后少了`-cli`)命令并按回车键执行了,将会安装 `react-native` 依赖包。很遗憾的是,这个包会破坏 `react-native` 的开发环境。这时,开发者可以通过“`npm uninstall -g react-native`”命令来删除所安装的内容,然后重新执行“`npm uninstall -g react-native-cli`”命令。

1.3 代码编辑环境搭建

React Native 没有带任何代码编辑工具,它的代码可以使用任何一款纯文本处理器处理编辑。笔者推荐使用 Sublime Text 3 编辑器进行代码编辑。

1.3.1 Sublime Text 3

如果你已经是 Sublime 用户,请确认 Sublime 是否安装有 JSFormat 插件。具体请看 1.3.2.4 节。

如果没有安装 Sublime,请访问 www.sublimetext.com/3 网址,下载 Sublime Text 3 的安装包。它会根据你使用的电脑是 Windows 还是苹果自动提供相应的安装包。

虽然目前 Sublime Text 3 还是测试版本,但请读者还是安装它而不是 Sublime Text 2。因为有些插件只支持 Sublime Text 3。

1.3.2 开发用插件

Sublime Text 3 安装好后,还需要安装一些插件来提升开发的便利性。首先需要安装插件包管理器

1.3.2.1 插件包管理器

打开 Sublime Text 3 软件,首先需要安装 Sublime Text 的 Package Control。访问 <https://packagecontrol.io/installation> 网址,按照提示,将代码拷贝到 Sublime 的控制台。按回车键后,将开始安装 Sublime Text 3 的 Package Control。安装后,需要重启 Sublime Text 3 软件。

重启后,PC: 按“`Ctrl+Shift+P`”快捷键; Mac: 按“`cmd+shift+P`”快捷键,出现功能选择面板,在功能选择输入框中输入 Package Control: Install package (输入前两个单词后,就可以在下面的选项中找到该项并直接选择),等待一小会儿,出现安装面板。

1.3.2.2 babel-sublime 插件

babel-sublime 插件支持 ES 6 语法与 React Native 的 JSX 语法在 Sublime Text 3 中的高亮显示,

便于开发者阅读。

安装方法是，在安装面板中输入 **babel**，然后选择出现的第一项，如图 1-2 所示。

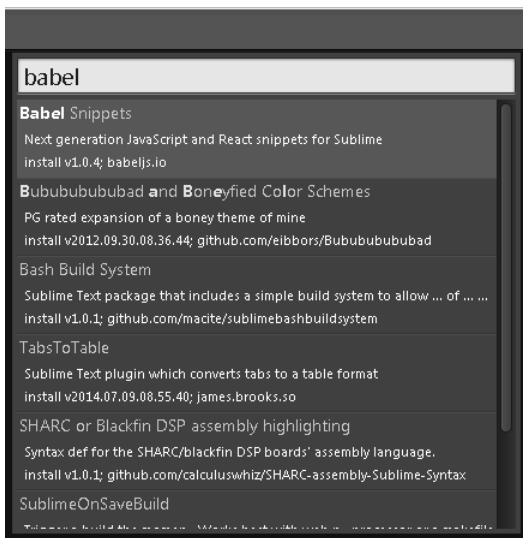


图 1-2 安装 babel-sublime 示意图

1.3.2.3 sublimeLinter 插件

sublimeLinter 是一个提供代码检测的插件工具，支持对 JavaScript 代码进行检测。

在笔者写本书时，React Native 开发正从使用 ES 5 语法转向使用 ES 6 语法。在使用 ES 5 语法编写 React Native 代码时，这个插件很好用。但目前 sublimeLinter 插件却会认为 ES 6 语法的代码是错误的，并且错误提示很烦人。因此在这个插件更新之前，不建议读者安装这个插件；或者安装后，暂时禁用它，直到它有了更新。禁用它的方法请参考 1.3.2.4 节。

安装方法是，在安装面板中输入 sublimeLinter，然后 Windows 用户以管理员身份进入命令行窗口，输入：

```
npm install -g JSHint
```

Mac 用户在命令行窗口输入：

```
sudo npm install -g HSHint
```

安装之后，需要配置 sublimeLinter。

Windows 用户打开 Sublime 后，选择 Sublime 主菜单中的“Tools”，然后选择“sublimeLinter”，再选择“Open user Settings”，这时在 Sublime 主编辑窗口中会打开一个文件。

开发者请访问：<https://github.com/xitaoque/1.3.2.3/tree/master> 中的配置文件以得到配置文件，将代码复制到 Sublime 主编辑窗口中。

注意代码的第 75 行、第 88 行与第 89 行，需要读者将对应的路径改为自己电脑的路径。

第 75 行为:

```
"windows": "C:\\Users\\Xitao\\AppData\\Roaming\\npm"
```

其中的 Xitao 是笔者用 Windows 电脑的用户名, 如果读者使用 Windows 电脑, 在配置时需要将它换为自己电脑的用户名。

如果使用 Mac 电脑, 则在命令行窗口输入:

```
which jshint
```

在命令行窗口中会显示 JSHint 的安装位置, 通常为/home/path/to/bin/jshint。

如果读者的 JSHint 安装位置与此不同, 则需要将第 74 行后一个值换为 JSHint 的安装位置。

第 88 行与第 89 行需要填写 Node 的安装位置。与上面的修改类似, 不再讨论。

1.3.2.4 JSFormat 插件

JSFormat 插件在用户保存代码时将自动格式化 JavaScript 代码。虽然它对 JavaScript 开发非常好用, 但对 React Native 开发却非常不好用, 因为无法识别用户输入的是 JSX 格式的代码, 会在保存代码时将部分 JSX 格式的代码调整得乱七八糟。因此, 如果在 Sublime 上安装了这个插件, 在使用 Sublime 进行 React Native 开发时, 需要禁用此插件。

禁用方法: 对于 PC, 按 “Ctrl+Shift+P” 快捷键; 对于 Mac, 按 “cmd+shift+P” 快捷键, 在出现的功能选择面板中输入 Package Control:Disable Package, 然后输入 JSFormat, 选中 JSFormat, 就可以完成禁用。

1.3.3 Sublime 界面风格选择

插件安装完成后, 选择合适的界面风格。Preference->Color Scheme->Babel 下的 Monokai Phoenix 风格很不错。

1.3.4 键盘使用习惯

如果你以前经常使用 Windows 操作系统的电脑, 现在把 PC 键盘接在 Mac mini 的 USB 口使用苹果操作系统, 那么会遇到几个操作很不习惯的地方。

1. 按 Windows 操作习惯使用拷贝、剪切与粘贴键

在苹果操作系统下用 PC 键盘, 拷贝快捷键是 “开始+C” 键, 其他两个键类推。修改这个比较简单, 进入系统偏好设置, 选择键盘, 出现如图 1-3 所示的 “键盘” 界面。

单击右下方的 “修饰键” 按钮, 在出现的窗口中, 在 “Control(^)键” 后的选择框中选择 “⌘Command”, 在 “Command(⌘)键” 后的选择框中选择 “^Control”。如图 1-4 所示是修改后的界面显示。

修改好后, 单击 “好” 按钮, 就可以按 Windows 操作系统的操作习惯来使用 “Ctrl+C”、

“Ctrl+V”、“Ctrl+X”快捷键了。



图 1-3 修改键盘使用习惯第一步



图 1-4 修改键盘使用习惯第二步

2. 按 Windows 操作习惯使用 Home 键与 End 键

在 Mac 操作系统下，Home 键的默认行为是让视图移动到文档首（注意光标位置并未移动），而 End 键的默认行为是让视图移动到文档末尾（同样，光标位置并未移动）。

如果你习惯了 Windows 操作系统的风格（按 Home 键将光标移动到当前行首，按 End 键将光标移动到当前行尾），则可以使用 KeyBindingsEditor 软件来修改。它对个人用户免费，其访问网址是：<http://www.cocoabits.com/KeyBindingsEditor/>。有兴趣的读者可以看一下它的介绍网页。下面简单来说一下它的用法。

- 下载 dmg 包。
- 打开 dmg 包，将里面的可执行文件拖出到任意位置并执行。
- 重启操作系统，修改完成，可以按 Windows 操作习惯使用 Home 键与 End 键了。

1.4 React Dev tool 安装

虽然 React Dev Tool 在后来的版本中支持 Firefox 浏览器，但目前对它的支持仍然不是很好，因此建议读者选择通过 Chrome 浏览器来运行 React Dev Tool。

考虑到 Google 网站国内无法访问的原因，所以不能用普通的方法安装 React Dev Tool。需要先国内其他网站下载 Chrome 浏览器独立版安装包，下载完成后运行安装。

使用任意浏览器打开网址：<https://github.com/facebook/react-devtools/releases>，打开后的页面顶部如图 1-5 所示。

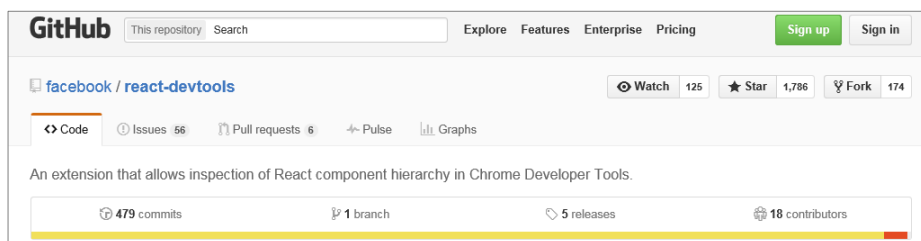


图 1-5 下载 React Dev tool

单击图中底部的“releases”，进入发布包下载页面。

下载最新版本的 react-devtools-x.xx.x.crx 文件，其中 x.xx.x 代表版本号。笔者写本书时，最新版本是 0.14.4。

打开 Chrome 浏览器，单击右上角的菜单按钮，在弹出的菜单中选择“设置”，然后在左侧边栏点击扩展程序。将下载的 crx 文件直接拖入 Chrome 窗口的空白处，这时会弹出确认框确认安装。安装完成后，在“允许访问文件网址”前打钩。

安装好后的界面如图 1-6 所示。

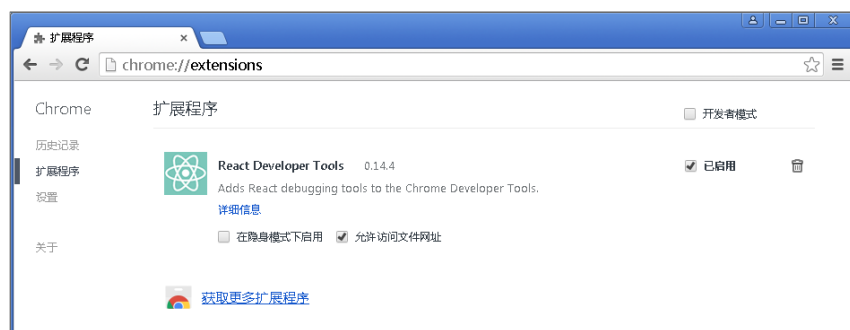


图 1-6 成功安装 React Dev tool

如果 Chrome 浏览器版本比较新，拖动不能安装的话，请使用下列步骤。

- (1) 将下载的文件后缀改名为.rar。
- (2) 将此 rar 文件解压缩到一个目录中，比如 react-devtools-0.14.4 目录中。
- (3) 删除此目录中的_metadata 子目录。

(4) 将此目录移动到一个合适的位置，比如用户目录下的文稿目录中（第 5 步操作执行后，这个目录位置不能再改变了）。

(5) 在 Chrome 浏览器的扩展程序页面中选择“加载已解压缩的扩展程序”，然后选择刚才解压缩出来的目录位置。

第 2 章

状态机思维与状态机变量

本章着重介绍 React Native 的状态机思维与实现它的基石:状态机变量。本章会使读者对 React Native 开发有一个大致的认识。

在本章中,我们将建立第一个项目。作为第一个项目,它的功能很简单,就是一个简单的注册页面,让用户输入手机号与密码,然后提供一个确定按钮供用户输入完成后点击。

2.1 初始化项目

打开命令行窗口,进入我们希望建立项目目录所在的父目录后,输入命令:

```
react-native init Project21
```

其中,Project21 是我们希望建立的项目名称,21 代表 React Native 当前是 0.21.0 版本。读者也可以将它换为任意名称。

问题: 在 init 之前怎么知道初始化的 React Native 项目是什么版本呢?

答案: 访问 <https://github.com/facebook/react-native/releases>, 网页中标出的 Latest release 版本就是你将要 init 的版本。

初始化项目时间视网速与网络状况而定。很多开发者反映初始化时间非常长,在这里给读者一个参考值。在 20Mbps 光纤上网的情况下,在 Windows 环境下使用 npm 从国外初始化一个工程项目(0.21.0 版本)只需要 4 分钟时间,需要下载 75MB 数据,包括 11991 个文件与 2286 个文件夹;而在 Mac 环境下需要近 10 分钟的时间,需要下载 148MB 数据,包括 61265 个文件与 6597 个子文件夹。在 Mac 环境下需要下载的文件数量多,是因为在 Mac 环境下初始化项目同时支持 Android 与 iOS 平台。

接下来需要升级 Android SDK 的编译文件(iOS 平台如果没有安装 Android 开发环境,则可以跳过这一步)。这个升级操作只需要在安装好 React Native 开发环境后,初始化第一个项目时执行一次,以后在初始化其他项目时不需要再次运行升级命令。

进入上面使用 init 命令建立的子目录下,在本例中,子目录名是 Project21,然后执行升级命令。需要输入命令如下:

```
cd Project21
react-native upgrade
```

按回车键后，开始升级，这个命令的执行时间很短。

提示：对于 Windows 用户，如果初始化项目出错，则可以尝试删除“系统安装盘符:\Users\用户名\.node-gyp”目录，然后再次执行初始化命令。

如果还有错误，则可以尝试删除“系统安装盘符:\Users\用户名\AppData\Roaming\npm-cache”目录。这个目录是加速缓存目录，删除不会引起任何错误。然后再次执行初始化命令。

在启动开发环境改写项目前，让我们看看 init 都做了些什么。

进入项目所在的目录下，输入“npm list”可以查看 init 命令都下载了哪些依赖包。

如果是在 Windows 下开发，那么进入 init 后生成的目录下，输入：

```
npm list >npmlist.txt
```

这条命令会在当前目录下生成一个 npmlist.txt 文本文件，用你所习惯使用的编辑器打开它查看内容。

如果是在 Mac 下开发，那么进入 init 后生成的目录下，输入：

```
npm list
```

因为 Mac 的命令行窗口可以保存非常多的显示信息，因此不需要像 Windows 那样把命令的输出转向到一个文本文件中，而是可以直接通过命令行窗口右侧的滚卷条向上下翻动查看。

在输出内容中列出了从网络侧都下载了哪些依赖包，它们各自的版本号是多少。需要提醒的一点是，React Native 在 init 时将项目需要用到的依赖包都下载到了当前目录下的 node_modules 目录中。如果这些依赖包有了新的版本，当前目录下的这些依赖包并不会自动更新。

在输出内容的最上方，会显示“react-native@x.xx.x”，它代表着 React Native 的版本号。

2.2 运行项目

在首次运行项目之前，建议读者对刚初始化的项目文件夹做一个备份。Mac 用户将鼠标指向项目文件夹后单击鼠标右键，选择“建立备份”；Windows 用户需要电脑上有 WinRAR 之类的备份软件，以对项目文件夹建立一个压缩包。在大部分情况下，我们不需要使用这个备份。但在讨论混合开发等高级开发知识时，会修改到目录层次非常深、非常多的项目文件（基本上都是编译环境自动修改）。当讨论完一个知识点，开始讨论下一个知识点时，最好能从备份中恢复出一个全新的项目文件夹。

项目初始化完成后，让我们来运行一下项目。下面分在 Android 平台调测与在 iOS 平台调测讨论如何运行项目。

2.2.1 使用 Android 手机进行调测

使用 Android 手机进行调测，是笔者推荐的调测方式。因为 Android 模拟器启动比较慢，而且 Android 手机的价格非常平易近人。React Native 支持使用 Android 模拟器进行调测，但笔者没有进行相关的探索。如果读者有需要，则请自行探索。

首先将手机与电脑通过数据线相连。

在命令行窗口输入“adb devices”命令。

如果手机上已经安装了正确的驱动程序并且打开了调试模式（这两项如何实现不在本书中讨论），则会看到类似于如下的显示：

```
List of devices attached
637c4a95      device
```

这表示手机已经准备好调测应用程序了。如果没有显示，则表示手机连接有问题，无法调测。

如果调测用的手机或者模拟器使用的是 Android 5.0 或者以上的操作系统，那么在命令行输入“adb reverse tcp:8081 tcp:8081”。执行这条命令，如果手机连接正常，则不会有任何显示。这条命令通过 adb 反向代理端口，将调试电脑的 8081 端口反向代理到测试机上。

上面两条命令因为经常用到，建议读者制作一个批处理文件（在 Windows 开发环境下）或者 shell 可执行脚本（在 mac 环境下）保存这两条命令，以及下面将要讨论的“react-native start”命令，以方便执行。

如果手机使用的是 Android 5.0 以下的版本，那么不需要执行上一条命令。现在需要保证手机与开发电脑使用同一个无线网络。

然后在命令行窗口输入“ipconfig”，查看为电脑分配的网络地址并记下来。

在安装目录下，输入命令：react-native run-android。

这条命令会编译刚初始化的项目，并且将编译好的安装包安装到手机或者模拟器中。在编译时，需要从网络侧下载某些编译依赖包。用时会稍长（如果网络条件不好，这一步有可能会不成功。如果这一步没有成功，建议找一个好用的 VPN 连接临时用一下）。编译依赖包只需要下载一次。将来如果再次运行这个命令以把应用安装到另一部 Android 手机上时，用时就会很短。

如果是在 Windows 环境下，手机在安装好之后会自动启动刚才安装的应用程序，但这时你看到的是红色的屏幕，因为电脑上的服务环境还没有启动。在命令行窗口输入“react-native start”命令，会显示如图 2-1 所示的信息。

在 Windows 环境下，“react-native start”执行的时间会比较长（在 0.21.0 版本优化后，时间是 1 分钟左右）。但在以后修改代码再调试时，就不需要这个过程了。

当开发者点击手机大红屏上的“Reload JavaScript”按钮，或者选择调试菜单中的“Reload JavaScript”菜单项，或者修改了 React Native 代码并保存后，调试服务器都会马上对代码进行重新编译，如图 2-2 所示。

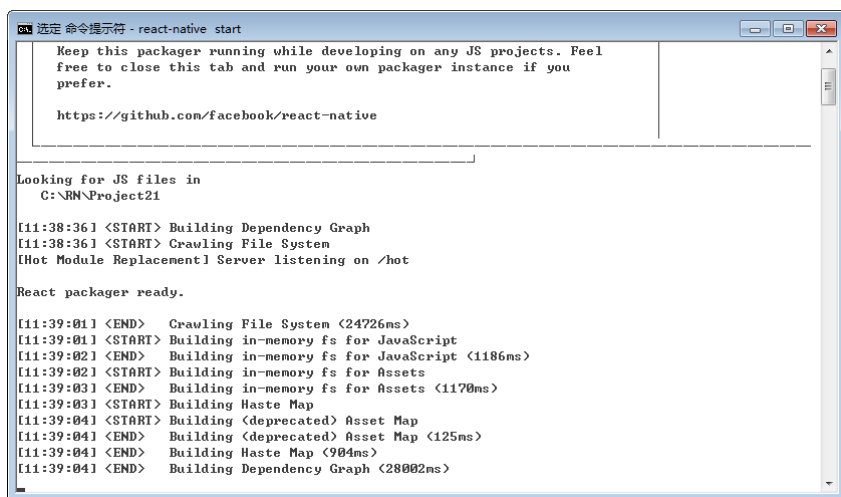


图 2-1 启动调试服务器显示信息

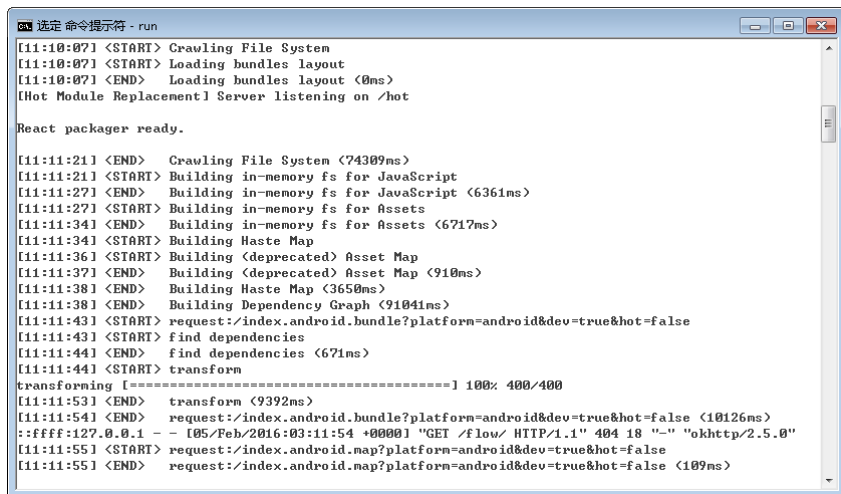


图 2-2 调试服务器重新编译显示信息

注意图 2-2 中所示的 transforming 行，行尾显示一个 400/400，这两个数字随着开发中使用不同的 React Native 组件而会有所不同，但关键是这两个数字要相等。如果不相等，则代表开发者的代码有错误。这时开发者可以在手机的大红屏上或者命令行窗口中的提示上看到哪行哪列的代码编译没有通过。

如果使用的 Android 手机操作系统版本低于 5.0，这时也会显示为红色屏幕。按 menu 键或者摇动手机，会在屏幕上弹出菜单，然后选择“Dev Settings”，再选择最下面的一项“Debug server host&port for device”。在弹出的输入框中输入刚才记录下的开发电脑的网络地址加上调试端口号。比如电脑的 IP 地址是 192.168.0.100，就输入 192.168.0.100:8081。点击“确定”，返回红色界面，再点击“Reload JavaScript”按钮，这时界面应当出来了。

总结一下：“react-native run-android”命令只在手机还没有安装项目时运行，然后就不需要再运行这个命令了，而是直接在手机界面上打开项目。

“react-native start”命令在每次调试应用程序前都需要运行，并且在调试中一直保持运行。

对于 Android 5.0 以下的手机，只需要设置一次调试用电脑的 IP 地址，它会自动保存。但读者**需要注意**：有可能调试用电脑的 IP 地址会在下一次启动后被更改，如果是这样，那么手机里的设置也需要做相应的更改。更好的办法是为电脑分配一个固定的 IP 地址（本书不讨论具体如何做）。

对于 Android 5.0 以上的手机，在每次手机与电脑连接后，准备调试应用程序前，都需要运行“adb reverse tcp:8081 tcp:8081”命令。将这条命令与“react-native start”命令做成一个批处理文件是一个比较不错的选择。

如果开发者的电脑以前做过其他的 JS 开发，则有可能在初始化项目后，项目无法正常运行。这时，需要删除刚初始化的项目目录。在命令行窗口输入“npm cache clean”，然后重新初始化项目。

下面介绍一下初始化项目运行时白屏的解决办法。

初始化得到的 Android 操作系统下的项目需要“显示悬浮窗”的运行权限，通常在使用“react-native run-android”命令将项目编译后得到的安装包安装到手机上时，手机会提示本应用需要“显示悬浮窗”权限，要求用户同意。但某些手机因为对 Android 操作系统做了修改，在开发模式下不会出现询问，导致程序安装后运行时白屏。这时要求开发者手动打开这个权限。

打开的方法大致如下（进入应用信息界面前的步骤，根据手机不同在文字提示上可能略有不同）：

选择 Android 手机的“设置”，然后选择“其他应用管理”，这时会出现一个应用列表。在列表中找到刚安装的应用名字（就是初始化项目的名称），点击应用，出现应用信息界面，如图 2-3 所示。

点击图 2-3 中的“权限管理”，出现应用权限管理界面，如图 2-4 所示。



图 2-3 应用信息界面



图 2-4 应用权限管理界面

在图 2-4 中，勾选“显示悬浮窗”权限。然后再次打开刚安装的应用，白屏问题就解决了。

2.2.2 使用 iPhone 手机或模拟器进行调测

使用 iPhone 手机或模拟器进行调测更简单。进入项目目录下的 ios 子目录中，点击 Xcode 工程项目文件，启动 Xcode。

启动起来后，在 Xcode 窗口的左上角有一个选择运行设备的选择按钮，点击它后，出现的界面如图 2-5 所示。

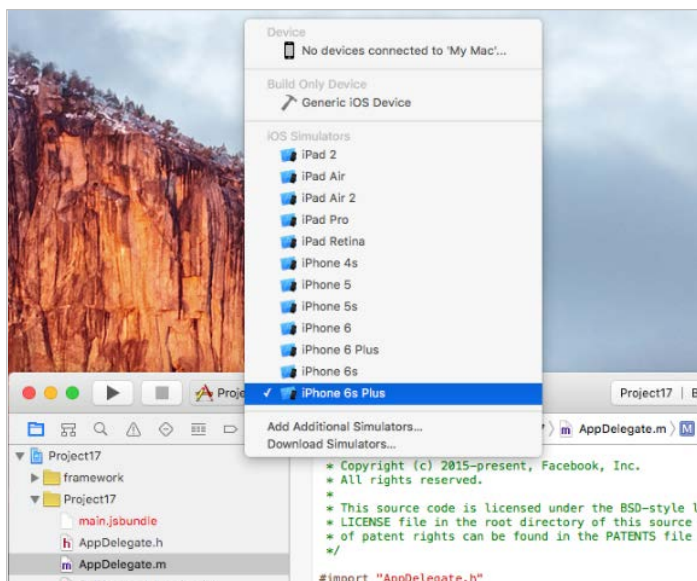


图 2-5 在 Xcode 中选择测试设备

如果连接了 iPhone 手机，图 2-5 所示列表中的第一个分栏就会列出连接的手机。在这个列表中上下移动鼠标选择开发者想要运行程序的设备。

如果选择模拟器作为应用程序的调测设备，那么现在就可以点击列表左侧的三角形按钮来编译项目，启动模拟器运行程序。在编译完成后，一个命令行窗口会自动弹出并运行。这个窗口就是调试服务器窗口，输出与图 2-1 非常类似。开发者可以最小化这个窗口不去理它，但不能关闭这个窗口。

选择 iPhone 手机作为调测设备稍微麻烦一些。首先开发者需要保证调测用的手机与开发者的苹果电脑在同一个 IP 子网下。说得通俗些，就是它们连接的是同一个无线路由器。

打开苹果操作系统的“系统偏好设置”，然后选择“网络”，会出现图 2-6 所示的窗口。

如果苹果电脑连接了 IP 网络，图 2-6 中状态描述的第一行将显示为“已连接”，第三行记录的是苹果电脑的 IP 地址。在笔者的苹果电脑上，它的 IP 地址是 192.168.2.103。

修改项目的 AppDelegate.m 文件，就是图 2-5 中左下角阴影突出的那个文件。



图 2-6 查看 Mac 电脑 IP 地址

打开 AppDelegate.m 文件，这个文件不长，其中有一行是：

```
JavaScriptCodeLocation = [NSURL URLWithString:@"http://localhost:8081/index.ios.bundle?platform=ios&dev=true"];
```

将其中的 localhost 改为开发者的苹果电脑的 IP 地址。修改完成保存后，就可以关闭刚才打开的网络窗口了。然后点击列表左侧的三角形按钮来编译、运行项目。在编译完成后，一个命令行窗口会自动弹出并运行。开发者可以最小化这个窗口不去理它，但不能关闭这个窗口。

如果运行后真机出现了一个大红屏，并提示：Could not connect to development server...，那么请确保你的 iPhone 连接上了无线网络，并且与苹果电脑连接的是同一个无线网络。然后检查刚才修改 localhost 时填写的苹果电脑的 IP 地址是否正确。

下面介绍一下 iOS 平台调试菜单。

Android 平台可以通过手机的 menu 键或者摇晃手机来弹出调试菜单。iOS 手机或者 PAD 可以通过摇晃手机或者 PAD 来弹出调试菜单。iOS 模拟器需要开发者按下电脑键盘上的“command+D”快捷键来弹出调试菜单。

开发者在使用 iOS 模拟器时，有时会不小心按了键盘上的“command+T”快捷键。这个快捷键会启动慢镜头动画调试功能——这是一个调试动画时非常有用的功能。不调试动画时，慢镜头动画功能会让开发者调试等待时间很长，难以忍受。再次按一下“command+T”快捷键，就可以关闭慢镜头动画调试功能了。

2.2.3 修改 JSX 代码

打开项目目录下的 index.android.js（如果使用 Android 手机调测）或者 index.ios.js（如果使用 iPhone 手机调测）文件。

我们对代码进行修改，修改文件内容如代码 2-1 所示。

代码 2-1:

```

'use strict';
import React, { //为了节约版面，将下面的多行写为一行，读者可以不修改
  AppRegistry, Component, StyleSheet, Text, View
} from 'react-native';
let Dimensions = require('Dimensions'); //请读者增加此行代码
let PixelRatio = require('PixelRatio'); //请读者增加此行代码
let totalWidth = Dimensions.get('window').width; //请读者增加此行代码
let totalHeight = Dimensions.get('window').height; //请读者增加此行代码
let pixelRatio = PixelRatio.get(); //请读者增加此行代码
class Project19 extends Component {
  render() {
    return (
      <View style={styles.container}>
        <Text style={styles.welcome}>
          pixelRatio={pixelRatio} //需要修改的第一行
        </Text>
        <Text style={styles.instructions}>
          totalHeight={totalHeight}; //需要修改的第二行
        </Text>
        <Text style={styles.instructions}>
          totalWidth={totalWidth} //需要修改的第三行
        </Text>
      </View>
    );
  }
}
const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#F5FCFF',
  },
  welcome: {
    fontSize: 20,
    textAlign: 'center',
    margin: 10,
  },
  instructions: {
    textAlign: 'center',
    color: '#333333',
    marginBottom: 5,
  },
});
AppRegistry.registerComponent('Project19', () => Project19);

```

在修改的代码中，我们首先通过 React Native 提供的 `require` 语句将 `Dimensions` 与 `PixelRatio` 模块加载至 `Dimensions` 与 `PixelRatio` 两个变量。然后声明了三个变量，即 `totalHeight`、`totalWidth` 与 `pixelRatio`。其中前两个变量分别保存屏幕的高与宽，其单位不是实际屏幕的物理像素，而是逻辑像素，在手机开发中，有时逻辑像素被称为 PT (Pixel Point)；而 `pixelRatio` 变量记录逻辑像素的密度，表示当前 React Native 应用中 1 PT 等于多少实际像素。比如密度为 3 时，React Native

里一个 1 PT 的点实际显示为 3×3 像素。`totalHeight` 与 `totalWidth` 变量如何使用在下一节中会介绍；`pixelRatio` 变量在后面的章节中介绍到显示图像时会用到。

在接下来的显示组件代码中，我们将例程中固定字符串替换为显示提示字符串及其对应的值。

当代码修改完成保存后，React Native 会检测到代码发生改变，并立即针对改变进行编译。使用命令行编译时显示类似于图 2-2。编译速度很快，开发者基本上不会有感觉。

通过手机的 menu 键（或者摇晃手机）弹出调试菜单，选择“Reload JavaScript”，手机的屏幕输出如图 2-7 所示（屏幕中部部分截图）。

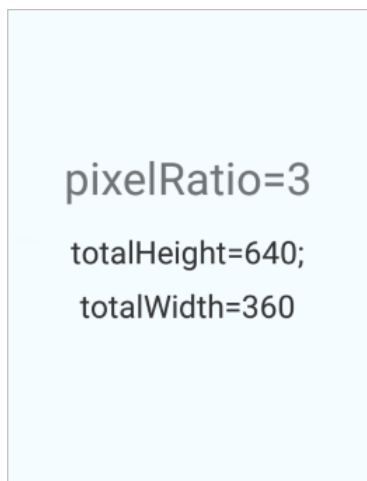


图 2-7 代码 2-1 运行输出结果

其中显示的具体数值因调测用的手机不同而可能会有所不同。

当我们修改了 JSX 代码后，只需要简单的一个“Reload JavaScript”，就可以马上看到对 UI 所做的变更，这就是 React Native 调试所带来的便利性。

2.2.4 ES 6 语法与 ES 5 语法

代码 2-1 是使用 ES 6 语法编写的 React Native 代码。在 React Native 刚出现时，使用 ES 5 语法编写代码。但随着 ES 6 的逐步完善，React Native 社区对使用 ES 6 语法开发也表现出了浓厚的兴趣。从 0.18.0 版本开始，React Native 初始化项目后的相当于 HelloWorld 例程的 `index.android.js` 与 `index.ios.js` 都使用 ES 6 语法编写。

对应代码 2-1 中的第 2~8 行，使用 ES 5 语法编写相应功能的代码如下：

```
let React = require('react-native');
let {
  AppRegistry,
  StyleSheet,
  Text,
  View
} = React;
```

对应代码 2-1 中的

```
class Project19 extends Component {
  .....
}
```

语句，使用 ES 5 语法编写相同功能的代码如下：

```
let Project19 = React.createClass({
  .....
}),
```

提示：class、extends 是 ES 6 的新语法特性。不熟悉这两个语法特性的读者请参阅附录 A.7。

ES 6 语法的写法与 Java 或者 C++ 语言的写法比较类似。因为 ES 6 是向下兼容 ES 5 的，所以使用 ES 5 语法开发的代码在以后新版本的 React Native 环境中编译、运行不会有问题。事实上，使用 ES 5 语法开发的代码可以用类似于一一对应的关系转为使用 ES 6 语法开发的代码。

目前 React Native 官方文档与网上开源的代码绝大部分还是使用 ES 5 语法编写的，因此本书中的讲解、例程也使用 ES 5 语法。从使用 ES 5 语法开发转为使用 ES 6 语法开发很容易。本书第 16 章详细讨论了如何从 ES 5 语法转为 ES 6 语法。读者学习完第 16 章之后，可以轻易地从使用 ES 5 语法开发转为使用 ES 6 语法开发。如果读者有需要提前掌握如何转向使用 ES 6 语法开发，那么在学习完第 2 章中讨论的知识后就可以直接去阅读第 16 章了。

使用 ES 6 语法开发的优点是可以利用一些 ES 6 语法新特性带来的便利。缺点是日前使用 ES 6 语法开发需要将每一个实例回调方法与实例绑定一次（这个绑定在 ES 5 中由 React 的 createClass 函数为开发者做了）。ES 5 语法的代码可以与 ES 6 语法的代码共存。第 16 章将详细讨论这些问题。

2.2.5 启动调试工具

让我们来熟悉一下 React Dev Tool 的调试功能。修改 Project19 组件的 render 函数如下：

代码 2-2：

```
.....
render: function() {
  let aValue;
  console.log('render has been executed.');
```

```
  console.log('totalHeight is:' + totalHeight);
  console.log('aValue is:' + aValue);
  console.log('the type of aValue is:' + typeof(aValue));
  return (
    .....
  )
}
```

在 render 函数的返回语句前加了 5 条语句。其中故意定义了一个没有初值的变量。然后通过 console.log 打印各种调试信息。

保存代码，然后在手机上“Reload JavaScript”。

调出调试菜单(Android 手机、模拟器按 menu 键, iPhone 模拟器按电脑键盘上的“command+D”快捷键, iPhone 真机调试请阅读 2.2.5.1 节), 选择“Debug in Chrome”菜单项。如果 iPhone 模拟器的弹出菜单对鼠标点击没有反应, 那么尝试按“Ctrl+Alt+回车”快捷键返回窗口模式, 稍等一会儿, 再点击菜单中的“Debug in Chrome”菜单项。

如果如第 1 章所述安装好了 React Dev Tool, Chrome 浏览器会自动启动。选择 Chrome 浏览器的 React Native Debugger 页面, 在 Windows 操作系统下按“Ctrl+Shift+J”快捷键(在苹果操作系统下按“command+option+J”快捷键), 显示页面上半部分类似于图 2-8(可以左右调整中线位置)。注意最下方显示的 4 条语句, 就是我们刚加入的语句打印的调试日志。我们可以通过调试工具, 将一些临时需要查看的变量的值、类型等信息打印在这里, 而不用调整手机 UI 组件来显示它们。这无疑提供了一种非常方便的手段。更多的调试功能将在本书后面继续介绍。

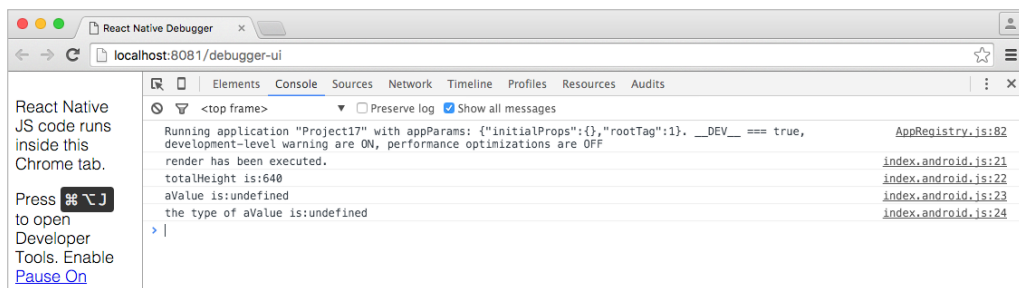


图 2-8 调试工具输出截图

在代码中, 我们除了可以使用 `console.log` 函数在调试工具中记录日志外, 还可以使用 `console.warn` 函数。这个函数的用法与 `log` 类似。与 `log` 相比, 它有两处不同:

- (1) 在调试工具的控制台中输出的字是醒目的红色。
- (2) 同时在手机屏幕上有黄色警告条。

`console.warn` 函数的显示效果, 请读者参阅本书第 3 章中的图 3-2 和图 3-3。

2.2.5.1 iPhone 真机调试配置

如果读者使用 iPhone 真机调试应用, 在调试过程中希望使用 React Dev Tool, 则需要在 Xcode 项目中找到并打开 `RCTWebSocketExecutor.m` 文件。它的位置如图 2-9 所示。

在文件中有如下语句:

```
NSString *URLString = [NSString stringWithFormat:@"http://localhost:%zd/debugger-proxy", port];
```

将语句中的 `localhost` 修改成 Mac 电脑的 IP 地址, 然后再次编译、运行项目。应用打开后摇晃手机以打开调试菜单, 选择“Debug in Chrome”菜单项。

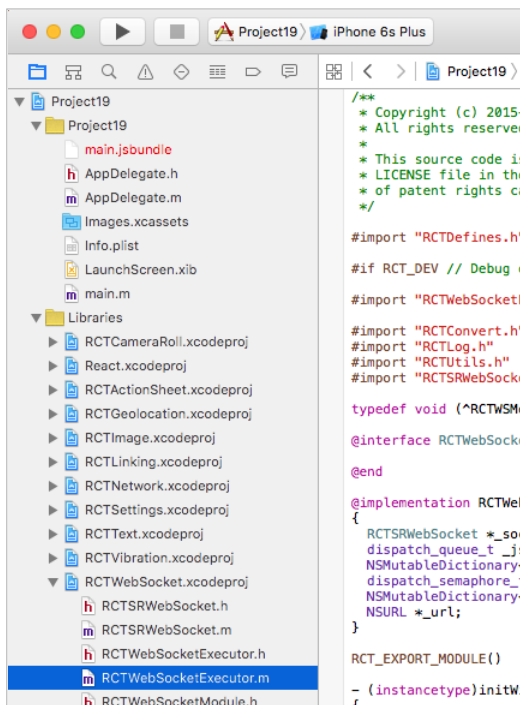


图 2-9 调试工具需要修改文件的位置

2.2.5.2 使用 iPhone 真机调试时开发环境的修改

从 React Native 0.18.0 版本开始,到笔者完稿时的 0.19.0 版本,使用 iPhone 真机调试时都会有一个 bug,就是调试工具的命令行栏会不停地出现警告信息,类似于图 2-10。

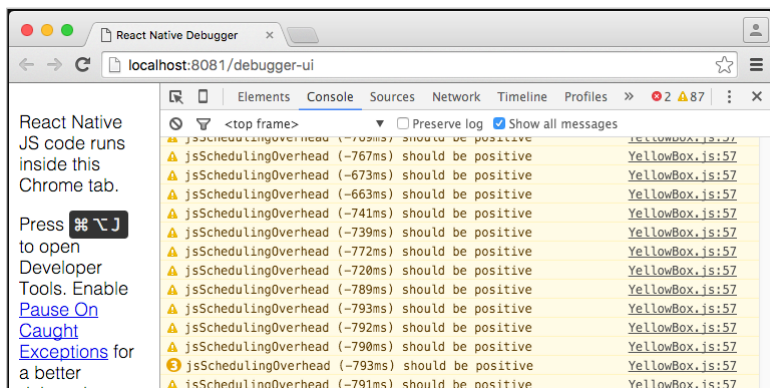


图 2-10 使用 iPhone 真机调试时的异常警告信息

因为警告信息不停地被打印,使得使用 iPhone 真机调试无法进行。这个问题估计是因为电脑的时间与手机的时间不一致,再加上处理的时延引起的。解决的办法有两个:

(1) 修改电脑或者手机的时间。但现在 Mac 电脑的时间是从苹果服务器获取的,手机的时间也通常是从移动运营商获取的,修改起来比较麻烦。

（2）修改代码。

修改代码是一个暂时的解决办法，希望 React Native 的下一个版本能解决这个问题。

打开项目文件夹下的 `node_modules/react-native/React/Modules/RCTTiming.m` 文件，在其中查找：

```
RCTLogWarn(@"jsSchedulingOverhead (%ims) should be positive",
(int)(jsSchedulingOverhead * 1000));
```

语句，在这条语句前加上 “//” 注释掉这条语句，然后保存退出。重新编译、运行项目，警告就会消失。

2.3 构建注册页面

现在我们来开发一个注册页面。如图 2-11 所示是这个注册页面上半部分效果截图（下半部分全是空白）。在这里展示注册页面效果，是为了先让读者对开发界面有一个大致的印象。为了不在一章中写入太多需要理解的知识点，这个页面设计得简洁、朴素。



图 2-11 注册页面

在这个页面中有 4 个区域。第一个用来提示并让用户输入手机号，第二个用来显示用户输入的手机号（这个区域完全是为了讲解 React 框架的 state 思维而出现的），第三个用来提示并让用户输入密码，第四个提供给用户一个“确定”按钮。

进入项目目录下，修改 `index.android.js` 文件（或者 `index.ios.js` 文件，如果你使用的是 iPhone 环境进行 React Native 学习），如代码 2-3 所示。因为读者可能是第一次接触 JSX 代码，所以在代码前加入了行号便于后面讲解。请读者在输入代码时，将每行行首的行号去掉。“//”后的部分是代码中的注释，读者不需要输入到代码中。

代码 2-3:

```
1      'use strict';
2      let React = require('react-native');
3      let {
4          AppRegistry, StyleSheet, Text, View, TextInput
5      } = React;
6      let Dimensions = require('Dimensions');
7      let totalWidth = Dimensions.get('window').width;
8      let leftStartPoint = totalWidth * 0.1;
9      let componentWidth = totalWidth * 0.8;
```

```

10 let Project19 = React.createClass({
11   render: function() {
12     return (
13       <View style={styles.container}>
14         <TextInput style={styles.numberInputStyle}
15           placeholder={'请输入手机号'}/>
16         <Text style={styles.textPromptStyle}>
17           您输入的手机号:
18         </Text>
19         <TextInput style={styles.passwordInputStyle}
20           placeholder={'请输入密码'}
21           password={true}/>
22         <Text style={styles.bigTextPrompt}>
23           确 定
24         </Text>
25       </View>
26     ); //这个反括号代表 render 函数的唯一一条语句即 return 语句的结束
27   }, //这个反大括号代表 render 函数定义的结束
28 }); //这个反大括号加反括号代表 let Project19 = React.createClass 语句的结束
29 let styles = StyleSheet.create({
30   container: {
31     flex: 1,
32     backgroundColor: 'white',
33   },
34   numberInputStyle: {
35     top: 20,
36     left: leftStartPoint,
37     width: componentWidth,
38     backgroundColor: 'gray',
39     fontSize: 20
40   },
41   textPromptStyle: {
42     top: 30,
43     left: leftStartPoint,
44     width: componentWidth,
45     fontSize: 20
46   },
47   passwordInputStyle: {
48     top: 50,
49     left: leftStartPoint,
50     width: componentWidth,
51     backgroundColor: 'gray',
52     fontSize: 20
53   },
54   bigTextPrompt: {
55     top: 70,
56     left: leftStartPoint,
57     width: componentWidth,
58     backgroundColor: 'gray',
59     color: 'white',
60     textAlign: 'center',
61     fontSize: 60
62   }
63 }); //这个反大括号加反括号代表 let styles = StyleSheet.create 语句的结束
64 AppRegistry.registerComponent('Project19', () => Project19);

```

提示：第 10 行的变量名 Project19，以及第 64 行的字符串常量“Project19”与其后的变量名 Project19，请读者更改为自己初始化项目时的项目名称。

第 1 行“use strict”表示 JavaScript 代码使用严格模式。第 2 行导入 React 组件。第 3~5 行是 ES 6 的解构赋值语句（不熟悉者可以参考附录 A.1）。为了节省本书版面，把它们由原来的每个一行改为了全部合为一行。解构赋值后，我们有了 5 个 JavaScript 变量：AppRegistry、StyleSheet、Text、View 和 TextInput。在这 5 个变量中，AppRegistry 与 StyleSheet 是 React Native 框架提供的 API（应用程序编程接口）；而 Text、View 与 TextInput 是 React Native 框架提供的组件。

提示：在 React Native 框架中 API 与组件的区别是，一个 React Native 组件是一个可显示的元素，通常被编写在它的父组件的 render 函数返回值中。当它被渲染时，手机 UI 界面会出现对应的显示区域。而 API 是 React Native 提供的类，开发者通常是在代码中调用它的静态函数以实现各种控制。比如代码 2-3 中的第 64 行，我们调用 AppRegistry 的静态函数 registerComponent 向 React Native 框架注册实现的 React Native 组件 Project19。

接下来，我们声明了 3 个变量，这些变量都是用来定义显示区域的某个显示指标的。大家应当注意到了，这 3 个变量都是动态计算的，它们的值都与获取到的屏幕宽度有关。

在代码运行时动态计算 UI 组件的宽、高是一个自适应手机屏幕的办法。但这个办法略为有些麻烦，并且对于宽、高动态变化的 UI 组件无能为力。在第 4 章中，我们将看到更好的布局方案。

leftStartPoint 定义组件从左侧起显示的起点位置。它的定义是屏幕总宽度的 10%。

componentWidth 定义组件的宽度。它的定义是屏幕总宽度的 80%。

第 10~28 行的分号从 JSX 语义上说是一条语句。这条语句调用 React 类的 createClass 函数定义并建立了名为 Project19 的对象，React 建立的对象通常都用于构建 UI 界面，每个组件负责手机屏幕上的一块区域。本书中将这种对象称之为 React Native 组件，以区别于其他与 UI 界面显示无关的对象。建立 Project19 组件的语句很长，不得不多行显示。

第 11~27 行定义了 Project19 组件的 render 函数。

从语义上说，第 12~26 行的 render 函数也只有一条语句，它直接 return 了一个 JSX 代码描述的 UI 组件。

第 13 行声明了一个 View，它的样式由 styles.container 定义。它是 UI 组件的根 View，有 4 个子组件，都在它的内部声明。

第 14~15、19~21 行声明了两个 TextInput 组件。第 16~18、22~24 行声明了两个 Text 组件。

第 25 行标识了根 View 组件的定义结束。

从语义上说，第 29~63 行也是一条语句，它定义了一个名为 styles 的对象。在 styles 内部定义了前面讨论的各组件样式。

第 64 行注册了我们所创建的 Project19 组件。

在第 13 行语句<View style={styles.container}>中, style={styles.container}表示给 View 组件指定一个属性。这个属性的名称是 style, 它的值是 styles.container。

向某个组件指定的属性, 在组件内部可以访问到。这也是父组件向子组件传递消息的方式。组件的属性在组件内部是一个常量, 它的值在组件内部是不可以改变的。

第 14~15 行声明了一个 TextInput 组件并给 TextInput 组件提供了两个属性。placeholder 用来定义用户未输入时在文本输入框中显示的字符串。

第 16~18 行声明了一个 Text 组件并提供了 style 属性。第 17 行定义了 Text 组件显示的字符串。

第 19~21 行声明了另一个 TextInput 组件并提供了 3 个属性。password={true}表示这个文本输入框用于输入密码。

第 22~24 行读者应当可以自行理解。

在利用组件拼接 UI 界面时, 除了声明需要哪些组件外, 还需要给组件提供相应的属性, 这样才能够正确地利用它们。其中最常用的一个就是 style, 用来定义组件如何显示。我们将在后续章节中详细讨论每一个组件需要使用以及可以使用哪些属性。

第 29~63 行中定义的 4 种样式, 大部分含义读者应当可以自行理解。这里解释几点。

1. flex:1

它表示组件的宽、高会动态扩展(详见第 5 章)。

2. top、left

它表示组件从父组件的顶端(左侧)向下(向右)多少位置显示, 单位是 2.2.3 节中讨论过的 PT。

3. width

它表示本组件的宽度为多少。单位同样是 PT。

4. 高度

在各组件的样式定义中, 都没有定义高度。父 View 因为设置了 flex:1, 它会占满全屏, 高度就是屏幕的总高度。而各子组件的高度没有明确设置, 则由组件的显示内容需要多高来决定组件的高度。

5. 颜色

颜色在 UI 中很重要。凡是涉及 color 的, 在开发阶段, 开发者都可以简单地赋给它英语单词中的各颜色值, 比如 pink、gray、white、green、red、blue、black、yellow 等。如果是正式项目, UI 设计师提供每一个显示区域的背景与图案 RGB 色值, 类似于'#F5FCFF', 开发者就可以直接把

这些值填入 React Native 代码中。React Native 还可以使用 RGBA 色值，我们在第 3 章会看到它的使用。

6. `textAlign: 'center'`

它表示文字显示的方式是居中显示。

7. `fontSize`

`fontSize` 的值决定了 `Text`、`TextInput` 组件中文字的大小，进而又决定了两个组件的高度。因为需要显示的区域少，在例程中，与高度有关的值都是直接给出数值。当需要显示的组件比较多时，为了做到自适应屏幕，字体的大小以及各组件的 `top` 都应当通过屏幕的高度动态计算出来。

代码输入完成后，请在 Android 手机上运行代码。如果希望在 iOS 手机或者模拟器上运行代码，那么在第 36 与 37 行之间，还有第 43 与 44 行之间插入一行新代码：

```
height: 30,
```

在 Android 平台上，React Native 的 `TextInput` 组件可以按照其字体大小来自动设置高度。但在 iOS 平台上，必须要在样式中设置高度；否则 `TextInput` 组件将不会显示。开发者可以简单地通过都设定高度的方法来避开平台的不同性。对于这个不同点，将在 3.1 节中介绍 React Native 如何实现代码、组件的平台自适应。

运行代码后，屏幕显示如图 2-11 所示的效果。读者可以试着在两个输入框中输入些字符。

在进入下一节前，读者可以尝试调整 `styles` 中的各项数值，看看数值调整后所显示的效果，并借此加深对 React Native 开发的初步理解。

2.4 React Native 代码执行逻辑

让我们再来反思一下在代码 2-3 中，开发者编写的代码都做了哪些事情。

首先，导入了一系列 React Native 组件与 API，并且声明了一些在组件中需要使用到的变量（第 29 ~ 63 行代码的本质也是声明了一个常量）。在第 10 ~ 28 行，开发者定义了一个组件（实质上是一个 JS 对象），并且在代码的第 64 行，通过 `AppRegistry` API 向 React Native 框架注册了所定义的组件。这就是开发者编写的代码所做的全部工作，也是开发者代码所需要做的全部工作。

剩下的工作将由 React Native 框架来完成。当一个组件通过 `AppRegistry` API 向 React Native 注册后，React Native 框架将把这个组件渲染到手机屏幕上，并在发生 UI 事件时，将相应的 UI 事件通知给组件。这样，开发者代码就能进行 UI 处理与相应的业务逻辑处理了。

读者应当不难猜测到：当 React Native 框架渲染一个组件到手机屏幕上时，会调用该组件的 `render` 函数，并按 `render` 函数返回的 JSX 代码描述的可渲染元素来渲染手机 UI 界面。

提示：事实上，注册后，React Native 框架所做的事情比将组件渲染到手机屏幕上要多很多。但在刚开始时，我们只说重点。组件注册后，React Native 框架为组件所做的完整的工作流程在本书 7.1 节中有详细的描述。

2.5 UI 框架工作基本机制

上一节我们已经得到了一个 UI 界面，如果你有一定的其他平台开发经验，此时一定在想：当用户点击“确定”按钮时，从手机号输入框中读取用户输入的手机号，从密码输入框中读取用户输入的密码，然后交给注册模块去处理。但 React Native 不是这样思维的。

2.5.1 状态机思维

React 框架将所有的 UI 视为一个简单的状态机，那么任意一个 UI 场景就是状态机中的一种状态。根据决定状态的状态机变量的值，React 框架渲染出状态机的当前状态——对于开发者来说，单个 UI 场景就被渲染出来了。随着状态机变量值的改变，UI 状态机也在不停地改变状态，UI 场景也随之不停地被重新渲染。这样一个过程可以很轻松地做到数据与 UI 保持一致。下面让我们修改代码一步步熟悉这个过程。将 Project19 组件的前半部分代码修改为代码 2-4。

代码 2-4:

```
let Project19 = React.createClass({
  getInitialState: function() {
    return {
      inputedNum: '',
      inputedPW: '',
    };
  },
  updateNum: function(newText) {
    this.setState((state) => {
      return {
        inputedNum: newText,
      };
    });
  },
  updatePW: function(newText) {
    this.setState(() => {
      return {
        inputedPW: newText,
      };
    });
  },
  render: function() {
    return (
      <View style={styles.container}>
        <TextInput style={styles.numberInputStyle}
          placeholder={'请输入手机号'}
          onChangeText={(newText) => this.updateNum(newText)} />
        <Text style={styles.textPromptStyle}>
          您输入的手机号: {this.state.inputedNum}
        </Text>
        <TextInput style={styles.passwordInputStyle}
          placeholder={'请输入密码'}
          password={true}
          onChangeText={(newText) => this.updatePW(newText)} />
        <Text style={styles.bigTextPrompt}>
          确 定
        </Text>
      </View>
    );
  }
});
```

```

        </Text>
      </View>
    );
  },
});

```

读者可能已经注意到，Project19 组件多了一个名为 `getInitialState` 的函数。每一个通过 `React.createClass` 建立的组件都可以有这个函数。如果有这个函数，当这个组件被初始化时，该函数将被执行。通常在这个函数中声明需要用到状态机变量。本例中，我们声明了两个状态机变量，它们的初值都是空字符串，通过名字应当知道它们对应保存的是用户输入的手机号与密码。

先看第一个 `Text` 组件，它的显示字符串的定义变成了“您输入的手机号：`{this.state.inputNum}`”。这个 `Text` 组件将在屏幕上显示“您输入的手机号：”这个字符串，然后紧跟着显示 `this.state.inputNum` 这个状态机变量的值。

再来看第一个 `TextInput` 组件。我们向它提供了一个 `onChangeText` 属性。在代码中，将一个函数赋值给了 `onChangeText`。

```
onChangeText={ (newText) => this.updateNum(newText) } />
```

这条语句对初学 JavaScript 的读者来说比较难看懂，所以在这里略花些篇幅分析一下。大花括号中是一个箭头符号定义的函数（不熟悉的读者可以参考附录 A.2）。它将收到的字符串为参数调用 Project19 组件的 `updateNum` 函数，并且将 `updateNum` 函数的返回值返回。

这个函数将在什么时候执行呢？它在每一次 `onChangeText` 事件发生时会被调用执行。

`onChangeText` 事件什么时候会发生呢？每当用户在输入框中输入一个字符或者按一次退格键时，`onChangeText` 事件都会被触发。

对于有经验的开发者，`onChangeText={ (newText) => this.updateNum(newText) } />` 语句可以改写成：

```
onChangeText={this.updataNum} />
```

这种写法，我们无法从代码上看到 `newText` 这个参数的传递过程，但它还是被传递了。它的可读性差了一些。

但是一定不要写成：

```
onChangeText={this.updataNum(newText) } />
```

这是初接触 ES 6 的开发者很容易忽视的一个地方。`{this.updataNum}` 是一个变量。大家还记得 ES 6 声明变量的 6 种方法吗？它们是 `var`、`function`、`let`、`const`、`import`、`class`。`this.updataNum` 是一个使用 `function` 关键字定义的变量，它指向我们定义好的函数，所以 `onPress={this.updataNum}` 能够正确地挂接上对应的处理函数。而 `{this.updataNum(newText)}` 呢？首先，它运行的结果不可知，因为 `newText` 没有定义。有箭头函数时，`newText` 是形式参数；没有箭头函数时，`newText` 就没有定义。其次，`{this.updataNum(newText)}` 是 `this.updataNum(newText)` 这条语句执行后的返回值。在本例中，因为 `updataNum` 函数没有返回值，它的值就是 `undefined`。将接口与 `undefined` 挂接上是明显不行的。在某些手机上，编译不会出错，但运行时会导致莫名其妙的白屏死机。因此这种错

误一定要注意！

最后，让我们看看 `updateNum` 函数。它通过调用 `this.setState` 函数（2.7.1 节将详细讨论这个函数）改变 `inputedNum` 这个状态机变量的值为 `newText`（这是用户输入的字符串），`updateNum` 函数不需要返回值。

让我们运行修改后的代码，然后在手机号输入框中随意输入点什么，再删除一、两个字符。

注意到了吗？每一次用户输入一个字符或者删除一个字符后，第二行的显示都与用户的输入同步。而我们在 `TextInput` 的定义以及 `updateNum` 函数中都没有显式地用代码修改第二行的显示。

读者也许会不解，为什么要监视用户的每一次输入呢？为什么不能在用户完成输入后一次性读取呢？因为这是获取用户输入最简单的办法。我们将在第 6 章中详细讨论 `TextInput` 组件。

读者也许会说，最简单的使用方式都这么麻烦啊！嗯，能简化，在 2.7.6 节中我们会看到怎么把它简化为一条语句。

在第二个 `TextInput` 组件声明中，`password={true}` 声明这个输入框用于密码输入。这样当用户输入下一个字符时，前一个字符会自动变为小圆点。

`updatePW` 函数基本与 `updateNum` 类似。细心的读者会发现，在它的 `this.setState` 语句中创建的函数相比 `updateNum` 少了一个参数。这是一种省略的写法，当我们不需要那个参数时，就可以不写。2.7.1 节将详细讨论 `setState` 函数。

在下一章中，我们将看到“确定”两个字是如何成为确定按钮的，本章暂不讨论，假设它是一个按钮。那么不论用户何时按下按钮，用户输入的手机号与密码都会存储在对应的状态机变量中。本章中，我们也不会实现对按钮事件的处理。假设用户注册的过程已经由原生代码实现了，那么我们需要做的就是将在 `React Native` 界面得到的用户输入交给原生代码，然后等待原生代码执行注册，再将注册结果通知 `React Native` 模块，`React Native` 模块通过手机屏幕将结果呈现给用户。我们将在下一章中学习相关技术。

2.5.2 “冒充常量”的状态机变量

在 `React Native` 代码中，我们可以通过“`this.state.状态机变量名`”来访问状态机变量。访问意味着可以读取变量的值，也可以改变变量的值。

在 `React Native` 开发中，开发者需要将状态机变量视为“不可变的常量”。在开发者的代码中，永远不要出现“`this.state.某状态机变量名 = 某值;`”这样的语句。这样的语句从 JavaScript 语法上来说说是合法的，并且可以被正确执行；但从 `React Native` 开发原则的角度来说，它是不合法的。当开发者需要改变状态机变量的值时，一定要并且只能使用 `this.setState` 函数（这个函数将在 2.7.1 节中详细讨论）。订立这个规则的原因之一参见 7.1.6 节。

当开发者需要可以直接改变的变量时，请定义 `React Native` 组件的成员变量，详见 2.8 节。

2.5.3 “无处安放”的状态机变量

当一个 React Native 组件有了第一个状态机变量后，这个组件就变成了有状态的 React Native 组件。一个没有状态的 React Native 组件也许会酸溜溜地对一个有状态的 React Native 组件说：“你是个有故事的人啊！”但对于 React Native 开发者来说，React Native 开发的一个原则是努力让自定义的 React Native 组件成为无状态的 React Native 组件，或者说在不影响程序结构的情况下，尽可能减少有状态的 React Native 组件的数目。

如果一个 React Native 组件的数据都是不变的(这也意味着它对应的 UI 显示部分也是不变的)，那么它就不应当成为一个单独的 React Native 组件，因为单独的 React Native 组件肯定有数据是会被改变的。对于无状态的 React Native 组件来说，会被改变的数据来自于它的 props（属性）；而对于有状态的 React Native 组件来说，会被改变的数据不仅来自于它的 props，还来自于它的 state（状态机变量）。尽可能让自定义的 React Native 组件成为无状态的 React Native 组件，意味着尽可能让自定义的 React Native 组件没有状态机变量。

哪些 React Native 组件必须有状态机变量呢？

当一个 React Native 组件需要处理用户的输入（包括文本输入、麦克输入与摄像头输入），或者需要处理网络侧发给应用程序的数据，或者需要处理定时器超时事件，或者需要处理自己订阅的事件消息等不可预知的输入型事件时，它就必须要有对应的状态机变量。

努力让用户的自定义 React Native 组件成为无状态的 React Native 组件，可以让状态机变量放在最合理的地方，减少冗余的代码，也让应用程序框架更清晰。

一个好的 React Native 设计思路是：创建多个只负责渲染数据的无状态 React Native 组件，将它们封装在一个有状态的 React Native 组件中，并把这个有状态的 React Native 组件的状态机变量的值通过 props 传给无状态的 React Native 组件(这时这些无状态的 React Native 组件是有状态的 React Native 组件的子组件)。在这种设计思路下，有状态的 React Native 组件封装了 UI 的交互逻辑，而无状态的 React Native 组件负责渲染 UI 界面。我们将在 7.5 节中详细讨论这个设计思路与实现步骤。

2.5.4 “努力瘦身”的状态机变量

状态机变量的改变会导致 React Native 组件的重新渲染(2.7 节中将详细讨论)，那么提高 React Native 应用程序性能的一种方法就是努力减少状态机变量的数目。

哪些变量才能入选作为状态机变量呢？

React Native 应用程序工作时，React Native 组件接收各种事件，对所接收到的事件的处理可能导致处理结果中的某些数据需要显示在 UI 界面上。这些数据可以成为该 React Native 组件的状态机变量。我们把它们称作状态机变量备选名单。

开发者需要对这份名单上的数据做进一步分析，找出重复的数据。重复的数据指：

- 该数据可以由备选名单上的其他数据通过某种规则计算得出；

- 该数据可以由组件属性中的数据通过某种规则计算得出；
- 该数据可以由备选名单上的其他数据再加上属性中的某些数据按某种规则计算得出。

把这些重复的数据踢出备选名单。这样，开发者就得到了一个状态机变量的最小集。

在 React Native 组件的 `render` 函数中，在正确的位置引用状态机变量。对于被踢出备选名单的数据，在它们原本应当存在的位置填写上计算出其值的方法。

这么做的优点是明显的，在状态机变量中保存多余的、与其他状态机变量有关系的状态机变量，意味着开发者需要时刻保证这些有关系的变量之间的同步，如果在开发、修改中的某一处未能保证这种同步（比如修改了一个变量的值，却没有修改另一个有关系的变量），则会导致 UI 显示错误。相比之下，只在状态机变量中保存最少的变量，其他有关系的数据都在 `render` 函数中写明这种关系的计算方法，然后由 React Native 框架来保证它们的同步，这样更不容易出错，并且需要开发的代码也最少。

最后提醒一点是：千万不要把一个 React Native 组件放在状态机变量中。正确的做法是把它放在 `render` 函数中，让它成为本 React Native 组件的子组件。这也是新手易犯的错误。

2.6 React Native 组件间通信

在代码 2-4 中，我们已经完整地实现了 React Native 父子组件之间的通信。在此总结一下。

父组件向子组件传递消息、数据通过对子组件的属性赋值来实现。比如前面讨论过的 `styles` 属性、`placeholder` 属性等。在第 3 章中，我们还会讨论如何自定义属性向自定义 React Native 组件传递消息。

子组件向父组件传递消息、数据通过回调父组件传递给自己的回调函数来实现。回调函数由父组件设置，被保存在子组件的某个属性中，等待需要向父组件传递消息的时机到来。比如前面讨论过的 `onChange` 属性与 `onChangeText` 属性。

除了在父子组件之间通信外，开发者在开发中还会有在无直接关系的 React Native 组件中实现消息传递的需求。本书将在 7.5.6.1 节中讨论实现这个需求的一种常用方法。

2.7 深入理解 UI 重新渲染的过程

当需要 React Native 重新渲染 UI 时，开发者使用最多的就是 `setState` 函数。除此之外，React Native 还提供了 `replaceState` 函数与 `forceUpdate` 函数。

2.7.1 合并状态机变量

React Native 框架使用 `setState` 函数来合并状态机变量。`setState` 函数的原型是：

```
setState(object oldState,function callback)
```

在 2.5 节中，我们讲到 `updateNum` 函数通过调用 `this.setState` 函数改变 `inputNum` 这个状态机

变量的值。现在让我们深入讨论一下这个过程。

开发者调用 React Native 组件的 `this.setState` 函数要求 React Native 框架重新渲染 UI 界面时，React Native 框架的回答是：“知道了，我会尽快渲染，但我首先要想一想如何才能高效地重新渲染界面，你把哪些状态机变量需要改变，以及需要改变成什么样子告诉我吧，我一会儿重新渲染界面时就按这些要求来做。”

所以，开发者传递给 `setState` 函数的是一个使用箭头符号定义的函数。当 React Native 想清楚了如何高效地渲染界面时，它会调用这个函数来查看哪些状态机变量需要被改变，并改变那些状态机变量，然后重新渲染界面。

为什么 React Native 需要想一想如何才能高效地重新渲染界面呢？因为一个组件的 UI 的可变部分不仅由它自己的状态机变量构成，还会由父组件传递进来的属性构成。在子组件来看，父组件传递进来的属性是不可改变的，但父组件可以改变它。如果父组件改变了它，那么子组件的 UI 就需要重新渲染。同时，本组件的状态机变量还有可能成为某子组件的属性，那么本组件的状态机变量的改变必然要求子组件的 UI 也要被渲染。在父子组件层层嵌套的结构下，React Native 需要计算哪一级的哪些部分需要重新渲染，而哪些部分可以保持原来的值等，如此才能做到高效地重新渲染界面。

当某子组件也需要重新渲染时，那么该子组件的各个生命周期函数都会被按生命周期规则调用。这也是 React Native 如此重视高效渲染的原因。生命周期函数将在第 7 章中详细讨论，在此读者只要明白这个原因就行。

看到这里，读者应当明白 React Native 可以帮助开发者分担多少复杂的工作。那么这里也是强调两个 React Native 重要开发守则的地方：

- (1) 只使用 `setState` 函数来改变状态机变量！
- (2) 尽一切可能让 UI 中可变的数据来源是状态机变量与属性！

再来看开发者传递给 `setState` 函数的那个使用箭头符号定义的函数，它可以有一个传入的参数。这个参数是什么？有什么用呢？我们来修改 `updateNum` 函数并加入一个新的函数 `changeNumDone`。修改后的代码片段见代码 2-5。在代码 2-5 中用到了 JavaScript 的 `for in` 循环语句，不清楚的读者可以查看附录 A.3。

代码 2-5：

```
1  updateNum: function(newText) {
2      this.setState((oldState) => {
3          for (var aName in oldState) {
4              console.log(aName);
5              console.log(oldState[aName]);
6          }
7          return {
8              inputNum: newText,
9              aBrandnewStateVariable: 'I am a new variable.'
10         };
11     }, this.changeNumDone);
```



```

12   },
13   changeNumDone: function() {
14     console.log('React Native has changed inputed Num');
15   },

```

重新加载代码，然后运行程序。在第一个输入框中先输入 9，再输入 6，执行后调试工作日志输出如下：

```

1   index.android.js:23 inputedNum
2   index.android.js:24
3   index.android.js:23 inputedPW
4   index.android.js:24
5   index.android.js:33 React Native has changed inputed Num
6   index.android.js:23 inputedNum
7   index.android.js:24 9
8   index.android.js:23 inputedPW
9   index.android.js:24
10  index.android.js:23 aBrandnewStateVariable
11  index.android.js:24 I am a new variable.
12  index.android.js:33 React Native has changed inputed Num

```

当在 `this.setState` 函数中使用箭头符号定义的函数被执行时，传递进去的参数是还没有被改变的当前所有的状态机变量（这是因为在这段代码里，开发者不能通过 `this.state` 变量名访问 `React Native` 组件的状态机变量）。代码 2-5 的第 3~6 行演示了如何遍历它们的键与值。开发者可以读取、处理这些值，然后再决定自己的新的状态机变量要怎么改变。

代码 2~5 的第 7~10 行定义了开发者希望如何改变状态机变量。注意，在这几行代码中，要求加入一个新的状态机变量。这在 `setState` 函数中是允许的。`setState` 函数将传入函数的返回值与当前状态机做一个合并操作，名称相同的状态机变量就用新状态机变量的值覆盖老的，有新增加的状态机变量就直接增加，原来状态机中存在的在本次设置状态机变量的值时没有赋予新值的状态机变量保持不变。

调试工作日志输出的第 10、11 行显示了新的状态机变量已经被添加，并且可以在第二次修改状态机变量时被遍历出来。

需要特别指出的是，如果 `setState` 函数将传入函数的返回值与当前状态机相比没有任何修改与增加，那么将不会进行渲染。

代码 2-5 的第 11 行说明还可以给 `this.setState` 函数提供第二个可选参数——一个回调函数，它将在 `setState` 完成并且重新渲染完成（如果需要的话）后被调用。

调试工作日志输出的第 5、12 行显示了回调函数被执行。

为什么需要这个回调函数呢？因为我们要求 `React Native` 修改变量并且重新渲染界面不在 `this.setState` 函数被调用后马上执行。如果还有些操作要求在界面重新渲染完成后进行，那么就不能放在调用 `this.setState` 的语句后，而是应当放在回调函数中。

通过 `setState` 函数修改状态机变量的值，所有与状态机变量的值有关系的组件都会被刷新。这个过程是 `React Native` 框架通过某些技术（我们将在 7.1 节中看到这些技术的一些描述）高效完成

的，可以节约开发者的大量精力与时间。这就是使用状态机变量的最大优点！我们的例程只有一个组件与状态机变量有关系。在复杂的情况下，我们可以定义一个上、下、左、右滑动的页面，页面上有列表、图表等各种各样复杂的组件，各个组件的显示与开发者定义的组件中的变量有各种各样的关系。这时因为变量的值改变而导致需要更新的 UI 组件就会非常复杂，React Native 的这个特性就能真正显示出其优越性了。

状态机变量是 React Native 中一个非常基本但非常重要的技术点。在这个例子中我们以这种非常简单的方式做了介绍，在后面还会看到它的更多应用。按 React Native 组件式开发的思维模式，任何 React Native 模块中动态的输入都应当设计为状态机变量。输入包括用户的输入、收到的网络消息中需要被显示的元素、定时器被触发事件处理中需要显示的元素等。

2.7.2 判断是否渲染

React Native 框架使用 `shouldComponentUpdate` 函数来判断接下来是否进行渲染。这个函数的原型是：

```
boolean shouldComponentUpdate(object nextProps, object nextState)
```

React Native 组件可以实现 `shouldComponentUpdate` 函数。如果实现了这个函数，当 React Native 决定重新渲染组件时，会先调用这个函数。如果这个函数返回 `false`，React Native 将放弃渲染组件。我们将在 7.1.6 节中详细讨论这个函数。

`shouldComponentUpdate` 会传入两个对象，分别代表接下来准备进行的渲染所基于的 `props` 与 `state`。遍历它们获取键与对应值的方法请参见 2.7.1 节的代码 2-5。

读者可以在例程中加入代码 2-6 中的 `shouldComponentUpdate` 函数，然后运行查看效果。

代码 2-6：

```
shouldComponentUpdate: function() {  
    if (this.state.inputNum.length < 3) return false;  
    return true;  
},
```

修改完代码，测试看到效果后，请把这节加入的代码删除，为进入下一章学习做好准备。

2.7.3 替换状态机变量

React Native 框架使用 `replaceState` 函数来替换状态机变量。`replaceState` 函数的原型是：

```
replaceState(object nextState, function callback)
```

不建议使用这个函数，因为 React Native 很有可能在将来的版本中去除这个函数，并且这个函数对从 `React.Component` 继承而来的 ES 6 类无效！

`replaceState` 函数的工作机制与 `setState` 函数类似，区别在于 `setState` 执行的是一个已存在的状态机变量与新状态机变量合并的过程；而 `replaceState` 函数会删除所有已存在的状态机变量。当 `replaceState` 函数执行完成后，只会剩下在该函数中提供的状态机变量。

提示：ES 6 语法不仅支持给一个对象动态增加成员变量，而且也支持动态删除一个对象的成员变量。`replaceState` 正是使用动态删除对象的成员变量来实现的。

动态删除对象的某个成员变量的代码示例如下：

```
.....
var aObject = {
  propertyA: 'aaa',
    propertyB: 'bbb',
};
delete aObject.propertyB;
.....
```

2.7.4 强制启动渲染

React Native 框架使用 `forceUpdate` 函数向开发者提供强制启动渲染能力。这个函数的原型是：

```
forceUpdate(function callback)
```

如果开发者因为某种原因，使得 UI 中可变数据的来源必须从状态机变量与属性外获取，那么可以使用 React Native 提供的 `forceUpdate` 函数，开发者通过这个函数要求 React Native 重新渲染界面。这个函数的调用将会导致所有级别的所有 UI 组件重新读取、计算与渲染，并且所有 UI 组件的生命周期函数都会按生命周期规则来执行。

调用 `forceUpdate` 函数导致的重新渲染过程，将不会调用 `shouldComponentUpdate` 来检查是否允许重新渲染！

开发者可以提供一个回调函数，这个回调函数将在渲染完成后被调用。

开发者应当尽可能避免使用此函数。

2.7.5 渲染过程

React Native 框架通过 `render` 函数实现重新渲染。该函数的原型是：

```
ReactComponent render( )
```

任何 React Native 组件都必须要有这个函数。它必须并且只能返回一个可渲染的组件描述。

开发者可以在自己的代码中调用这个函数以重新刷新组件。但这不是一个好办法，正确的办法是修改某组件的状态机变量或者属性（对于一个组件来说，它的属性是常量，但对于给予该组件属性的父组件来说，子组件的属性就是它的状态机变量或者普通成员变量，可以随意修改），由此修改触发 React Native 框架的重新渲染过程。

2.7.6 合并状态机变量的最简语法

在完全明白合并状态机变量调用的 `setState` 函数的完整调用格式后，我们来讨论一下它的最简单格式。毕竟，怎么简单怎么来这个思想已经深入了 JavaScript 开发的骨髓。

代码 2-7 是例程中用户每更改输入号码一个字符后更改状态机变量中保存密码变量的函数。在原来的写法中，`newText` 是一个字符串，是用户更改后的密码字符串。

代码 2-7:

```
updateNum: function(newText) {
    this.setState(() => {
        return {
            inputNum: newText,
        };
    });
},
```

现在我们首先把原来的 `newText` 这个形式参数的名字直接更换为与状态机变量中保存密码变量的变量名相同的名字：`inputNum`。更改形式参数的名字不会产生任何影响，只是更换了名字。换名之后，合并状态机变量的简化语法如代码 2-8 所示，最终简化语法如代码 2-9 所示。

代码 2-8:

```
updatePW: function(inputNum) {
    this.setState(()=>
        return {inputNum};
    );
},
```

代码 2-9:

```
updatePW: function(inputNum) {
    this.setState({inputNum});
},
```

而 2.5.1 节中很复杂的改变状态机变量的写法也可以简化为：

```
onChangeText={({inputNum}) => this.setState({inputNum})}/>
```

代码 2-8、2-9 与代码 2-7 完全等价。虽然这些写法可以让代码更简洁，但对于初学者会产生错误的印象：这条语句立即就执行了，容易忘记 `setState` 是异步执行的函数，以及其他等 `setState` 可以做到的事。因此本书后面章节还会不时地使用完全格式的写法。

2.8 React Native 组件的成员变量

其实我们在前面已经定义并使用了 React Native 组件的成员变量，但新接触 React Native 的读者可能还没意识到，因此再讨论一下。在 JavaScript 中，函数也是一种变量。因此前面我们创建的 React Native 组件的成员函数就是 React Native 组件的成员变量。

在 React Native 组件中，与本组件显示有关的变量存放在状态机变量中，父组件传递下来的属性存放在属性变量中。如果在开发中还需要一些与组件逻辑控制相关但与组件显示无关的变量，比如需要在组件被挂接时申请并保存一些资源，在组件被卸载时释放；又比如订阅一些事件与取消订阅。这些变量保存在哪里呢？

可以保存在状态机变量中，但是强烈建议开发者不要这么做。在状态机变量中存放与显示无关的变量，会导致 React Native 无必要地判断是否需要重新渲染 UI，从而导致应用性能下降。正

确的做法是保存在 React Native 组件的成员变量中。

React Native 组件的成员变量定义很简单。看代码 2-10 示例。

代码 2-10:

```
.....
let Project19 = React.createClass({
  _myProperty1: 'test',
  _myProperty2: true,
  getInitialState: function() {
    .....
  }
});
```

示例代码的第 3、4 行为 Project19 组件定义了两个成员变量，它们的名字是：_myProperty1、_myProperty2。这样，在组件的各个成员函数中，开发者通过 this._myProperty1、this._myProperty2 访问或者修改这两个成员变量。

React Native 组件的成员变量名字没有强制要求以下画线开头，但以下画线开头是一个良好的习惯。通过这种方式可以明确地提醒开发者，这是一个组件的成员变量。

2.9 React Native 组件的静态变量、静态函数

React Native 允许组件有静态变量。开发者通过“组件名.变量名”的方式可以访问、修改一个组件的静态变量。

在 JavaScript 中，函数也是一种变量，所以开发者可以为 React Native 组件定义静态函数。代码 2-11 中定义了一个静态变量和一个静态函数，并调用了静态函数。请读者注意其调用语法。

代码 2-11:

```
.....
statics: {
  _myStaticObject: 'init value',           //定义类的静态成员变量
  myStaticMethod: function() {           //定义类的静态成员函数
    console.log('myStaticMethod is called.')
  }
},
render: function() {
  console.log(Project19._myStaticObject); //访问类的静态成员变量
  Project19.myStaticMethod();             //调用类的静态成员函数
}
.....
```

修改完代码，测试看到效果后，请把这节加入的代码删除，为进入下一章学习做好准备。

第 3 章

页面导航、弹出框及深入理解属性

在第 2 章中，我们建立了一个注册页面。现在让我们整体考虑一下这个为了学习 React Native 而设计的应用程序。当用户打开程序后，首先进入的是注册页面。用户填写了手机号码与密码后，应当进入一个等待注册结果的页面，当原生代码在后台完成注册后，把注册操作的结果通过 React Native 实现的 UI 通知给用户。因此，我们需要三个页面，分别是填写信息、等待注册结果和注册结果页面。这三个页面有着逻辑上的联系，React Native 通过 Navigator 组件可以轻松地实现页面之间的导航。

3.1 分离注册组件、组件平台自适应

3.1.1 分离注册组件

第 2 章实现的注册页面是保存在 `index.android.js` 文件（或者 `index.ios.js`）中的，让我们将它分离成一个单独的组件。

首先，在 Project19 工作目录中，复制一份 `index.android.js`（或者 `index.ios.js`）文件，并将复制的文件更名为 `RegisterLeaf.js`。

然后，打开 `RegisterLeaf.js`，将原来的“`let Project19 = React.createClass({`”语句改为：

```
let RegisterLeaf = React.createClass({
```

再将最后一条 `AppRegistry` 语句去掉，在这个位置增加：

```
module.exports = RegisterLeaf;
```

这条语句声明 `RegisterLeaf` 组件已经准备好供外部其他模块调用了。

3.1.2 组件平台自适应

在 2.3 节的最后，我们提到了 `TextInput` 组件在 Android 平台上可以不设置组件高度，而在 iOS 平台上必须要设置组件高度。本节利用这个微小的不同点介绍如何让代码自适应平台。

现在注册页面已经被实现为一个 React Native 组件，代码存放在单独的与 React Native 组件名同名的 `js` 文件中，我们需要让这个 React Native 组件能平台自适应。在项目文件夹下，对

RegisterLeaf.js 文件做一个备份。然后将备份文件改名为 RegisterLeaf.ios.js，将 RegisterLeaf.js 文件改名为 RegisterLeaf.android.js。最后，让 RegisterLeaf.android.js 文件中 TextInput 组件的样式设置中没有高度那一行（在 2.3 节末尾添加的那一行），让 RegisterLeaf.ios.js 文件中 TextInput 组件的样式设置中有高度那一行。组件的平台自适应就这样完成了。

当我们启动项目时，React Native 框架会自动判断项目运行在哪一个平台下，然后自动加载各个组件针对不同平台的源代码文件。

3.1.3 平台检测

在开发中，有些时候还是要根据当前代码运行的不同平台而执行不同的代码。也许分支代码不多，开发者不希望按 3.1.2 节中的自适应规则生成两个代码文件。这时开发者可以通过 React Native 提供的 Platform API 来进行判断。判断代码如下：

```
.....
let Platform = require('Platform');
if (Platform.OS === "android") {
    .....//Android 平台需要运行的代码
} else {
    .....//iOS 平台需要运行的代码
}
.....
```

本书 7.5.3 节中代码 7-18 示范了如何在渲染界面的同一份 JSX 代码里按不同平台渲染不同界面。

在接下来的学习中，读者会看到很多组件都有针对某平台的特殊属性。这些属性只在某一个平台上有效；如果在另一个平台上运行的代码中有这个属性，什么异常也不会发生，就好像代码中没有这个属性一样。

3.2 导航组件、挂接注册组件

打开 index.android.js，修改代码，如代码 3-1 所示。

代码 3-1：

```
'use strict';
var React = require('react-native');
var {
    AppRegistry,    Navigator,    BackAndroid
} = React;
var RegisterLeaf = require('./RegisterLeaf');
var WaitingLeaf = require('./WaitingLeaf');

var NaviModule = React.createClass({
    configureScene: function(route) {
        return Navigator.SceneConfigs.FadeAndroid;
    },

    renderScene: function(router, navigator) {
        this._navigator = navigator;
```

```

    switch (router.name) {
      case "register":
        return <RegisterLeaf navigator={navigator}/>;
      case "waiting":
        return <WaitingLeaf phoneNumber={router.phoneNumber}
          userPW={router.userPW} navigator={navigator} />
    }
  },
  componentDidMount: function() {
    var navigator = this._navigator;
    BackAndroid.addEventListener('NaviModuleListener', () => {
      if (navigator && navigator.getCurrentRoutes().length > 1) {
        navigator.pop();
        return true;
      }
      return false;
    });
  },
  componentWillUnmount: function() {
    BackAndroid.removeEventListener('NaviModuleListener');
  },
  render: function() {
    return (
      <Navigator
        initialRoute={{name: 'register'}}
        configureScene={this.configureScene}
        renderScene={this.renderScene} />
    );
  }
});
AppRegistry.registerComponent('Project19', () => NaviModule);

```

在导航模块中，首先声明了需要用到两个 API：AppRegistry 与 BackAndroid，还有一个 React Native 组件 Navigator。注意，新的 Project19 组件不需要使用 StyleSheet API，我们就可以不声明它。

接下来，我们导入了在 RegisterLeaf.js 文件中提供的 RegisterLeaf 模块，以及马上要实现的 WaitingLeaf 模块。两条 require 语句直接生成了两个 React Native 组件。

通过 React 模块的 createClass 函数，我们建立 NaviModule 组件。首先看 configureScene 函数，这个函数的用途是告知 Navigator 模块我们希望在视图转换时使用何种效果。使用这个名字是特意要与 Navigator 组件的 configureScene 函数同名。在定义与挂接时，可以清楚地知道这个函数的用途。

renderScene 函数用来告知 Navigator 模块我们希望如何挂接当前的视图。函数名也特意与 Navigator 组件的函数同名。在这个函数中，我们声明当页面导航路径的 name 为 register 时，将导入在第 2 章中实现的 RegisterLeaf 模块挂接到当前的视图；而当页面导航路径的 name 为 waiting 时，将导入 WaitingLeaf 模块。我们将在后面详细描述导入的过程。

componentDidMount 和 componentWillUnmount 函数是 React 框架的两个生命周期函数。NaviModule 组件实现了这两个函数后，当这个组件被挂接到当前页面或者被移除时，这两个函数

会被调用。

在 NaviModule 模块的 componentDidMount 函数中做了两件事：一是声明了一个名为 navigator 的变量并对其进行赋值；二是设置了一个对 Android 手机后退事件的监听器。

监听器发现 Android 手机的后退键被按下时，会检查当前 navigator 变量是否存在并且它的当前导航路径长度是否大于 1。如果大于 1，则表示当前视图下至少还有一个视图。用户按下后退键被理解为希望返回上一个视图，因此返回上一个视图，并且返回 true 值，表示后退事件已经被处理。

componentWillUnmount 函数在组件被移除前，清除组件被挂接时设置的后退事件监听器。

NaviModule 组件的渲染函数声明它只有一个组件：Navigator。我们为这个组件初始化了三个属性，其中第一个是初始导航视图的名称；后两个已经在前面描述了。

最后，我们注册 NaviModule 模块。注意，注册的名称还是第 2 章使用的 Project19，但注册的箭头函数返回值已经换为 NaviModule 了。

如果将 `var WaitingLeaf = require('./WaitingLeaf')` 语句注释掉，代码就已经可以运行了。运行后读者可以发现，手机屏幕上显示的效果与第 2 章结束时的成果没有区别。在 NaviModule 中，某些代码要求使用 WaitingLeaf 这个目前还没有实现的模块。但这并不影响我们现在就运行它，只要处理流程还没走到用到 WaitingLeaf 的地方，就不会报错。这也是 ES 6 与 Java、Objective-C 之间很大的一个区别。

3.3 挂接注册等待组件

在应用程序的处理流程中，当用户按下确定键（如何使用图像生成美观的按钮将在后续章节中介绍）开始注册时，应用程序跳转到注册页面。为了实现这个功能，修改 RegisterLeaf.js 文件的部分代码，如代码 3-2 所示。

代码 3-2:

```
.....
    <Text style={styles.bigTextPrompt}
        onPress={this.userPressConfirm}>
        确 定
    </Text>
</View>
);
},
userPressConfirm: function() {
    this.props.navigator.push({
        phoneNumber: this.state.inputNum,
        userPW: this.state.inputPW,
        name: 'waiting',
    });
},
});
.....
```

提示：在第 5 章中我们将介绍如何实现各种精美好看的按钮。因为使用 Text 组件做一个简单的按钮代码简单，有助于开发者把注意力放在应当关注的代码上，所以本书中会大量使用这种简单但不怎么美观的 Text 组件按钮。

首先，我们在 Text 组件中对 onPress 事件进行了挂接。当这个 Text 组件被按下时，RegisterLeaf 组件的 userPressConfirm 函数将被执行。

然后，在 userPressConfirm 函数中，我们调用了 RegisterLeaf 组件的 navigator 属性的 push 函数。

在 push 函数中，我们传递了一个类的实例，它有三个成员，分别是 phoneNUM、userPW 和 name。这些成员都是字符串变量并且被相应地赋值。

当 navigator 的 push 函数被调用后，通过 React Native 的 Navigator 组件的工作机制，NaviModule 的 renderScene 函数将被调用，并且 push 函数传入的变量成为了 renderScene 函数的第一个参数。注意在 renderScene 函数中的处理：

```
case "waiting":
  return <WaitingLeaf phoneNumber={router.phoneNumber}
    userPW={router.userPW} navigator={navigator} />
```

当 name 等于 waiting 时，将 WaitingLeaf 模块挂接上去。在这条语句中，phoneNumber、userPW 和 navigator 三个属性被同时传入。

现在让我们看看 WaitingLeaf 模块的实现。WaitingLeaf.js 文件的内容如代码 3-3 所示。

代码 3-3:

```
'use strict';
let React = require('react-native');
let {
  StyleSheet, Text, View,
} = React;
let WaitingLeaf = React.createClass({
  render: function() {
    return (
      <View style={styles.container}>
        <Text style={styles.textPromptStyle} >
          注册使用手机号: {this.props.phoneNumber}
        </Text>
        <Text style={styles.textPromptStyle}>
          注册使用密码: {this.props.userPW}
        </Text>
        <Text style={styles.bigTextPrompt}
          onPress={this.goBack}>
          返回
        </Text>
      </View>
    );
  },
  goBack: function() {
    this.props.navigator.push({
```

```

        name: "register"
      });
    },
  });
let styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#F5FCFF',
  },
  textPromptStyle: {
    fontSize: 20
  },
  bigTextPrompt: {
    width: 300,
    backgroundColor: 'gray',
    color: 'white',
    textAlign: 'center',
    fontSize: 60
  }
});
module.exports = WaitingLeaf;

```

在渲染函数中,我们用两行文字显示了传入的两个属性,然后声明了下一行文字是可点击的。当这行文字被点击时, `goBack` 函数被执行。

完成 `WaitingLeaf.js` 的编辑后,执行整个程序。输入手机号,再输入几个字符作为密码,然后直接点击下面的确定按钮。

点击确定按钮后,进入等待页面,可以看到手机号和密码被正确地显示出来。

正式商用发布的移动应用不会在等待登录时把用户输入的密码显示在等待界面上,本章的示例只是用于功能展示。

在本节最后,对属性做一个总结。属性用于 `React Native` 组件外界向组件内传递变量或者数据。通过 `this.props` 属性来访问。当属性的取值在组件外界被改变时,如果属性被组件中的 UI 以某种方式显示出来,则显示也会相应地发生改变。在 `React Native` 组件内部,我们不可以对属性进行赋值操作,这种做法会导致运行时出错。

使用 `React.createClass` 函数建立的组件在被挂接时,可以通过属性向组件内传递变量或者数据。那么这些属性何时生效呢?我们将在讨论 `React Native` 组件生命周期时解释这个问题。目前只要知道当 `render` 函数(这个函数是任何 `React Native` 组件都必须有的)被调用后,我们就可以正确地得到属性的值即可。

3.4 Navigator 组件工作机制

本节只是初步讨论 `Navigator` 组件的部分工作机制。关于 `Navigator` 组件的完整工作机制,将在第 10 章中详细讨论。

3.4.1 push 与 pop

现在请按返回键，注意每按一次的变化，直到退出程序。请注意，需要按 4 次才能退出程序。下面我们仔细描述一下整个过程。

当 NaviModule 模块被初始化时，因为 initialRoute 指向了 register，Navigator 组件通过 renderScene 函数查到需要显示 RegisterLeaf，于是拿出一张纸，在纸上画出了 RegisterLeaf，放在手机屏幕上。

当我们点击确定按钮时，Navigator 组件再拿出一张纸，画出了 WaitingLeaf，放在了手机屏幕上。

当我们点击文字时，goBack 函数被执行，Navigator 组件再拿出一张纸，画出了 RegisterLeaf，放在手机屏幕上。

当我们再点击确定按钮时，Navigator 组件又拿出一张纸，画出了 WaitingLeaf，放在了手机屏幕上。

然后我们按下返回键。因为 NaviModule 监听了这个事件，因此相应的处理函数被调用。在处理函数中，Navigator 看了下，在屏幕上它画的纸大于一张(navigator.getCurrentRoutes().length > 1)，于是它就把最上面的一张纸撕下来，扔进垃圾堆(navigator.pop();)。

于是，下面的那张原来被遮盖住的纸露了出来。

直到 Navigator 撕到发现只有一张它画的纸时，它说，“这个返回事件我不能处理了”，于是处理函数返回 false，声明这个返回事件没有被处理。那么 Android 系统处理了这个返回事件后，也就结束了应用程序(iOS 因为没有返回键，所以没有返回键处理逻辑)。

在这里，我们使用了 Navigator 的 push 与 pop 函数，其优点是 push 后，原来的组件还存在内存中。也就是说，当我们在等待页面退回登录页面时，原来在登录页面的输入还存在。

3.4.2 replace 函数

开发者还可以使用 Navigator 组件的 replace 函数。push 函数可以想象成 Navigator 组件在原来的屏幕上新放了一张纸，那么 replace 函数就可以想象成 Navigator 组件把原来屏幕上的纸拿下来扔了，然后放了一张纸。使用 replace 函数替代 push 与 pop，可以销毁不需要使用到的组件，节省应用内存开销。

我们可以把 RegisterLeaf 模块中的跳转页面改为如代码 3-4 所示。

代码 3-4:

```
userConfirmed: function() {
  this.props.navigator.replace({
    phoneNumber: this.state.inputNum,
    userPW: this.state.inputPW,
    name: 'waiting',
  });
},
```

再将 WaitingLeaf 模块中的跳转页面改为如代码 3-5 所示。

代码 3-5:

```
goBack: function() {
  this.props.navigator.replace({
    name: "register"
  });
},
```

修改之后的导航机制与原来的实现有两点差异：

- (1) 在等待页面无法通过 Android 手机的返回键返回到登录页面。
- (2) 在等待页面通过点击“退回登录页面”回到登录页面后，用户原来的输入都被丢弃。

3.5 自定义组件

到目前为止，我们都是利用 React Native 提供的基础组件对它们进行排列组合的，就像搭积木一样。现在让我们来看一下如何实现自定义组件，以及如何嵌套使用自定义组件。

如果读者看过 React Native 官网文档上的基本组件目录，也许会发现一个问题：React Native 还没有提供弹出菜单之类的基本 UI 组件。

为什么 React Native 没有提供呢？React Native 的回答很简单：需要时你自己做一个，想做啥样就做啥样。使用 React Native 框架，可以很方便、快捷地生成各种格式的组件。

因为本章还不会讨论图片组件、可触摸组件的使用，所以实现的自定义组件还是比较朴素的。但利用本章内容与下一章内容，开发者应当可以开发出实用美观、各种式样的基本组件。

3.5.1 “弹出一切框”的实现

在这里，我们假设用户按了确定键后，在手机屏幕上弹出一个询问框，要求用户确认是否登录，并给用户提供“确定”、“取消”两个按钮。

敏锐的读者可能会想到，本章讨论的 Navigator 组件可以实现类似功能，只要再加一个确认页面就可以了。我们确实可以通过这种机制来实现。但这么实现有一个问题，那就是确认页面是一个新的界面，它完全遮盖住了原来的界面，即使将新的界面背景颜色设为透明色也没用。而我们希望实现的效果是，当弹出询问框时，原来的界面不可触摸操作，没有被弹出框遮盖住的部分只是变暗一些，但仍然能够看到（效果见图 3-1）。

从 React Native 0.17.0 版本起，Alert API 开始支持 Android 平台，这个 API 从原来的只支持 iOS 平台变为跨平台支持。本节介绍的弹出框完全可以通过 Alert API 来实现（我们将在 3.9 节中详



图 3-1 弹出框效果图

细讨论 Alert API)。但 Alert API 有其局限性，表现为：

- 弹出框始终显示在屏幕的中间位置（水平、垂直方向都居中）；
- 在 Android 平台上，弹出框的可选项最多只能有三个；
- 按钮的排列方式固定，不能调整位置。

本节讨论的技术可以克服上面的局限性，并且稍做修改就可以用来实现任何弹出性质的组件，比如弹出输入框、弹出菜单或者任何格式的自定义弹出 React Native 组件。同时，通过本节这个短小精悍的例子，可以很好地引出必须要介绍的基本概念。

在项目目录下建立 ConfirmDialog.js 文件，其代码如代码 3-6 所示。

代码 3-6，ConfirmDialog.js:

```
1  'use strict';
2  let React = require('react-native');
3  let Dimensions = require('Dimensions');
4  let totalWidth = Dimensions.get('window').width;
5  let totalHeight = Dimensions.get('window').height;
6  let {
7    StyleSheet, Text, View, BackAndroid,
8  } = React;
9  let ConfirmDialog = React.createClass({
10    render: function() {
11      return (
12        <View style={styles.confirmCont} >
13          <View style={styles.dialogStyle}>
14            <Text style={styles.textPrompt}>
15              {this.props.promptToUser}
16            </Text>
17            <Text style={styles.yesButton}
18              onPress={this.props.userConfirmed}
19              numberOfLines={3}>
20              {'\r\n'}确 定
21            </Text>
22            <Text style={styles.cancelButton}
23              onPress={this.props.userCanceled}
24              numberOfLines={3}>
25              {'\r\n'}取 消
26            </Text>
27          </View>
28        </View>
29      );
30    }
31  });
32  let styles = StyleSheet.create({
33    confirmCont: {
34      position: 'absolute',
35      top: 0,
36      width: totalWidth,
37      height: totalHeight,
38      backgroundColor: 'rgba(52,52,52,0.5)'
39    },
40    dialogStyle: {
```

```

41     position: 'absolute',
42     top: totalHeight * 0.4,
43     left: totalWidth / 10,
44     width: totalWidth * 0.8,
45     height: totalHeight * 0.3,
46     backgroundColor: 'white'
47   },
48   textPrompt: {
49     position: 'absolute',
50     top: 10,
51     left: 10,
52     fontSize: 20,
53     color: 'black'
54   },
55   yesButton: {
56     position: 'absolute',
57     bottom: 10,
58     left: 10,
59     width: totalWidth * 0.35,
60     height: totalHeight * 0.12,
61     backgroundColor: 'grey',
62     fontSize: 20,
63     color: 'white',
64     textAlign: 'center'
65   },
66   cancelButton: {
67     position: 'absolute',
68     bottom: 10,
69     right: 10,
70     width: totalWidth * 0.35,
71     height: totalHeight * 0.12,
72     backgroundColor: 'grey',
73     fontSize: 20,
74     color: 'white',
75     textAlign: 'center'
76   }
77 });
78 module.exports = ConfirmDialog;

```

代码 3-6 的第 18 行直接将 Text 组件的 onPress 事件与父组件传递进来的 userConfirmed 属性挂接。第 19 行声明 Text 组件可以显示三行。第 20 行声明 Text 组件的显示内容是一个回车换行再跟着“确定”。第 23~25 行与之类似。这里对 Text 组件进行这样特殊显示，是为了克服 Text 组件不能垂直居中显示。我们将在第 5 章中详细讨论这个问题。

第 33~39 行，confirmCont 定义了 ConfirmDialog 的根 View 的样式。position: 'absolute'，声明这个 View 的位置使用绝对定位；top:0，定义绝对定位的起点，所以从屏幕的 (0,0) 坐标显示起，然后宽、高都是总宽与总高。换句话说，这个根 View 全屏显示。

3.5.2 React Native 中颜色类型的值

在 React Native 开发中，经常会说某个属性是颜色类型。严格来说，颜色类型不是一种单独的

类型，它也是一个字符串，只是用来描述一种颜色。因此在讨论时也经常使用颜色类型。

在简单的开发调试中，开发者经常使用'white'、'grey'、'black'、'red'、'green'这种英文颜色单词。

从 React Native 0.20.0 开始，要求在颜色类型值上放弃使用以前比较混乱的方法。除了使用简单的颜色单词外，开发者还可以使用 RGBA、RGB、HSL、HSLA 格式或者数值来描述颜色。

confirmCont 的背景颜色：'rgba(52,52,52,0.5)'。这是一个 RGBA 色值的描述，美工在设计界面时可能会交给你不同的 RGBA 色值，而开发者就可以用这种方式使用它。最后一个值的取值范围是 0~1，表示颜色的透明程度。有兴趣的读者可以试着修改这个数值，看看不同的效果。这个全屏的 View，当它被挂接上时，会遮盖住原来屏幕上的其他组件。因为它是半透明的，因此用户可以看到被遮盖住的组件，但却无法对被遮盖住的组件进行任何操作。

RGBA 颜色还可以使用如“0xF5FCFF01”格式表示，其中 0x 表示它是一个十六进制数，其后的每两个数分别对应 R、G、B、A 的值。

“#F5FCFF”、“rgb(245,252,255)”都是 RGB 格式的颜色值。前者是十六进制表示；后者是十进制表示。前者“#”号后的固定 6 个十六进制数，每两个数分别对应 R、G、B 的值；后者逗号分隔的数值的取值范围是 0~255。RGB 是 Alpha 值（透明度）为 1（完全不透明）的 RGBA 值。

HSL 与 HSLA 格式的颜色描述是 hsl(h,s,l)和 hsla(h,s,l,a)，例如 hsl(360, 100%, 100%)、hsla(360, 100%, 100%, 1.0)。

如果读者看不懂这些颜色格式，也没有关系，因为这些颜色格式数据都是由美工提供给开发者的，开发者只需要把它们用在代码中就可以了。

代码 3-6 的第 42、43 行定义了 dialogStyle 描述的起点位置。因为 dialogStyle 也是绝对定位的，这个起点位置是相对于它的父 View 的最上行与最左侧的。

其他样式声明的含义请参考前面章节中的解释。

至此，一个简单的弹出确认框组件就写好了。当在其他代码中需要弹出确认框时，直接导入这个组件就可以了。

3.5.3 挂接自定义组件

下面在 RegisterLeaf 中使用我们刚开发的 ConfirmDialog。

在 RegisterLeaf.js 文件头部加入：

```
let ConfirmDialog = require('./ConfirmDialog');
```

在 getInitialState 函数中加入新的成员变量 needToConfirm，它的初值是 false。修改后的 getInitialState 函数见代码 3-7。

代码 3-7：

```
getInitialState: function() {  
  return {
```



```

    inputedNum: '',
    inputedPW: '',
    needToConfirm: false
  };
},

```

请按代码 3-8 为 RegisterLeaf 组件修改 userPressConfirme 函数，并增加 userCanceled 函数和 userConfirmed 函数。

代码 3-8:

```

userPressConfirme: function() {
  this.setState((state) => {
    return {
      needToConfirm: true
    };
  });
},
userCanceled: function() {
  this.setState((state) => {
    return {
      needToConfirm: false,
    };
  });
},
userConfirmed: function() {
  this.setState((state) => {
    return {
      needToConfirm: false,
    };
  });
  this.props.navigator.replace({
    phoneNumber: this.state.inputedNum,
    userPW: this.state.inputedPW,
    name: 'waiting',
  });
},

```

当用户点击确定按钮时,新的 userPressConfirme 函数将状态机变量 needToConfirm 修改为 true,并要求 React Native 框架重新渲染界面。

userCanceled 函数定义了当用户点击弹出询问框中的“取消”按钮后如何处理。

userConfirmed 函数定义了当用户点击弹出询问框中的“确定”按钮后如何处理。

修改原来的 render 函数,在函数头与 return 语句间加入如下代码。当状态机变量 needToConfirm 的值为真时,返回 renderWithDialog 函数的返回值。

```
if (this.state.needToConfirm) return this.renderWithDialog();
```

然后为 RegisterLeaf 组件加入新的 renderWithDialog 函数,见代码 3-9。

代码 3-9:

```
renderWithDialog: function() {
```

```

    console.log('renderWithDialog');
    return (
      <View style={styles.container}>
        <TextInput style={styles.numberInputStyle}
          placeholder='请输入手机号'
          onChange={this.updateNum}/>
        <Text style={styles.textPromptStyle}>
          您输入的手机号: {this.state.inputNum}
        </Text>
        <TextInput style={styles.passwordInputStyle}
          placeholder='请输入密码'
          password={true}
          onChangeText={(newText) => this.updatePW(newText)}/>
        <Text style={styles.bigTextPrompt}
          onPress={this.userConfirmed}>
          确 定
        </Text>
        <ConfirmDialog userConfirmed={this.userConfirmed}
          userCanceled={this.userCanceled}
          promptToUser='使用'+this.state.inputNum+'号码登录? ' />
      </View>
    );
  },

```

新定义的 `renderWithDialog` 函数与原来 `render` 函数返回的前面部分是一样的，只是在最后加入了：

```

<ConfirmDialog userConfirmed={this.userConfirmed}
  userCanceled={this.userCanceled}
  promptToUser='使用'+this.state.inputNum+'号码登录? ' />

```

这一部分我们声明了需要使用 `ConfirmDialog` 这个自定义组件，并且给这个组件传入了三个属性。通过 `userConfirmed` 挂接上了对用户点击“确定”按钮的处理，通过 `userCanceled` 挂接上了对用户点击“取消”按钮的处理，`promptToUser` 传入了给用户展示的文字提示。

通过将子组件的属性初始化为父组件的某个函数，打通了子组件向父组件通信的通道。在这里，这个通道很简单，就是一个无参数的函数。但在需要的情况下，子组件可以通过有参数的函数向父组件传递数据。这是 React Native 开发中经常使用的基本技术。

代码修改完成后，运行代码，将显示如图 3-1 所示的界面。

3.6 BackAndroid API 的 bug 与解决办法

现在弹出框能正常出现在界面上，用户点击“确定”或者“取消”按钮也都能够正确处理。但是，当弹出框出现后，Android 手机用户按返回键时，将直接退出程序，而不是取消弹出框。这不符合使用习惯。

为了修改这个不便之处，我们需要让 `ConfirmDialog` 组件在被挂接上时能够监听并处理返回键被按下事件，而 `ConfirmDialog` 组件在要被取消时停止监听处理。在 3.2 节中，我们看到 React Native 的 `BackAndroid` API 可以提供这个功能。

BackAndroid API 的工作机制是，当挂接多个 Listener 后，用户按下返回键时，多个 Listener 都会监听到返回键被按下事件，并且它们的处理函数都会被执行。执行的顺序就是添加的顺序，不会因为前面的处理函数处理了返回事件，后面的处理函数就不会被执行了。这些处理函数都执行完后，只要有一个处理函数返回了 `true`（表示返回键被按下事件已经被处理），返回键被按下事件就不会交给 Android 框架处理，也就是说，应用程序没有办法退出。如果大家有兴趣，可以自行通过 React Native Dev tool 打印调试信息来观察其工作机制。

非常遗憾的是，直到 React Native 0.16.0 版本，BackAndroid API 还存在着 bug。目前 BackAndroid 的 `removeEventListener` 还不能正常工作，开发者添加的监听器无法被卸载。我们可以使用同一个标识字符串来添加多个监听器，每个监听器中的处理函数都会被执行。这也可以自行通过 React Native Dev tool 打印调试信息来观察其工作机制。

这个 bug 反映到我们所讨论的项目上，问题就是如果在 `ConfirmDialog` 中增加了监听器，这个监听器会始终工作。在我们无法改变这个事实的情况下，只能去适应这个事实。因此，我们需要 `ConfirmDialog` 始终知道它何时应该拦截返回键事件，何时不应该拦截返回键事件。

在 `RegisterLeaf` 组件中再增加一个函数，见代码 3-10。

代码 3-10:

```
tellConfirmDialogItsStatus: function() {
  return this.state.needToConfirm;
},
```

在挂接 `ConfirmDialog` 时再增加一个属性 `amStillAlive`，见代码 3-11。

代码 3-11:

```
<ConfirmDialog userConfirmed={this.userConfirmed}
  userCanceled={this.userCanceled}
  amStillAlive={this.tellConfirmDialogItsStatus}
  promptToUser={'使用'+this.state.inputNum+'号码登录?'} />
```

在 `ConfirmDialog` 组件中增加两个函数，见代码 3-12。

代码 3-12:

```
componentDidMount: function() {
  var amStillAlive = this.props.amStillAlive;
  BackAndroid.addEventListener('ConfirmDialogListener', () => {
    if ( amStillAlive() ) {
      this.props.userCanceled();
      return true;
    }
    return false;
  });
},
componentWillUnmount: function() {
  BackAndroid.removeEventListener('ConfirmDialogListener');
},
```

虽然我们已经知道了目前 `removeEventListener` 无法正常工作，但还是在实现上调用了它。一

旦 React Native 开发小组修复了这个 bug，我们的代码就可以释放掉不需要的资源。

注意监听处理函数中判断条件的取值，我们没有直接使用 `this.props.amStillAlive()`，而是先用一个变量获取属性的值，然后再将这个变量值传入。这看起来有点多余？

事实上，我们必须这么做。因为监听处理函数在 `ConfirmDialog` 中只是描述了它的实现，当它真正被执行时，执行的上下文并不在 `ConfirmDialog` 中。也就是说，当它被执行时，如果运行 `this.props.amStillAlive()` 语句，它会发现找不到 `this.props.amStillAlive` 这个变量。而 `amStillAlive` 却是可以找到的，并且可以通过这个函数调用 `RegisterLeaf` 中的相应函数得到正确的条件值。

至此，一个可以正常工作的弹出框就实现完成了。

3.7 属性确认

为了正确地使用刚实现的弹出框自定义组件，需要给它提供四个属性，三个回调函数和一个字符串。因为自定义组件是可以复用的，项目组的其他同事可能会将我们刚写的 `ConfirmDialog.js` 文件使用到其代码中。但是该同事是个很粗心的人，没有问清楚用法，也没有仔细阅读代码，就想当然地使用上了，结果是怎么用也不对。

在开发 React Native 自定义组件时，可以通过属性确认来声明这个组件需要哪些属性。这样，如果在调用这个自定义组件时没有提供相应的属性，则会在手机与调试工具中弹出警告信息，告知开发者该组件需要哪些属性。属性需求声明的实现见代码 3-13。

代码 3-13:

```
propTypes: {
  userConfirmed: React.PropTypes.func.isRequired,
  userCanceled: React.PropTypes.func.isRequired,
  amStillAlive: React.PropTypes.func.isRequired,
  promptToUser:
    React.PropTypes.string.isRequired,
},
```

现在我们在 `RegisterLeaf` 组件调用 `ConfirmDialog` 组件的地方将所有传递进去的属性都删除，再运行代码，运行结果如图 3-2 所示，在屏幕下方弹出了 4 个警告。



图 3-2 属性确认警告

如果此时运行 React Native Dev Tool，则会显示如图 3-3 所示。

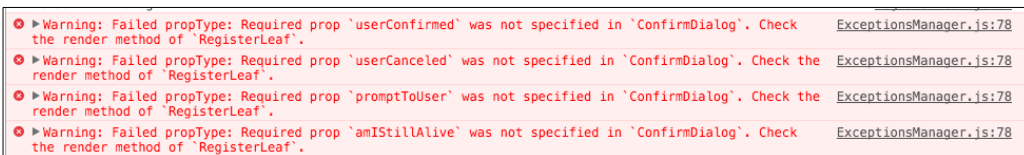


图 3-3 日志记录中的属性确认警告

为了保证 React Native 代码的高效运行，属性确认仅在开发环境中有效。也就是说，正式发布的 APP 运行时是不会进行检查的。下面让我们看看属性确认的语法。

1. 要求属性是指定的 JavaScript 基本类型

```
属性名称: React.PropTypes.array,
属性名称: React.PropTypes.bool,
属性名称: React.PropTypes.func,
属性名称: React.PropTypes.number,
属性名称: React.PropTypes.object,
属性名称: React.PropTypes.string,
```

2. 要求属性是可渲染节点

可渲染节点指数字、字符串、数字数组、字符串数组。

```
属性名称: React.PropTypes.node,
```

3. 要求属性是某个 React 元素

```
属性名称: React.PropTypes.element,
```

4. 要求属性是某个指定类的实例

```
属性名称: React.PropTypes.instanceOf (NameOfAClass),
```

5. 要求属性取值为特定的几个值

```
属性名称: React.PropTypes.oneOf(['值 1', '值 2']),
```

6. 属性可以为指定类型中的任意一个

```
属性名称: React.PropTypes.oneOfType([
  React.PropTypes.string,
  React.PropTypes.number,
  React.PropTypes.instanceOf (NameOfAClass)
]),
```

7. 要求属性为指定类型的数组

```
属性名称: React.PropTypes.arrayOf (React.PropTypes.number),
```

8. 要求属性是一个有特定成员变量的对象

```
属性名称: React.PropTypes.objectOf (React.PropTypes.number),
```

上面的语句要求传入的对象有一个成员变量是 number 类型。

9. 要求属性是一个指定构成方式的对象

```
属性名称: React.PropTypes.shape({
  color: React.PropTypes.string,
  fontSize: React.PropTypes.number
}),
```

10. 属性可以是任意类型

属性名称: `React.PropTypes.any`,

上面描述的 10 种语法, 都可以通过在后面加上 `isRequired` 声明它是必需的。

3.8 指定属性默认值

我们可以在自定义组件中设定属性默认值, 这样当组件被渲染时, 如果没有指定某个属性的值, 则使用默认值。通过在自定义组件中增加 `getDefaultProps` 函数来为属性指定默认值。下面我们为弹出框的提示字符串增加一个默认值。增加函数见代码 3-14。

代码 3-14:

```
getDefaultProps: function() {
  return {
    promptToUser: '你确定吗?',
  };
},
```

同时记得要将指定 `promptToUser` 为必需的“`isRequired`”去掉。

现在, 读者可以去除提供给 `ConfirmDialog` 的 `promptToUser` 属性, 这样默认值就会生效了。

3.9 Alert 应用程序编程接口

弹出确认框与弹出询问框是移动应用程序开发中经常需要使用的 UI 手段。在 3.5 节中, 我们自己开发实现了一个弹出框。本节将讨论 React Native 为我们准备好的弹出确认框和弹出询问框。

React Native 为开发者提供了 Alert API (应用程序编程接口), 供用户灵活地实现弹出确认框和弹出询问框。

通过 Alert API 显示的弹出框, 将始终显示在屏幕的中间位置。

在使用 Alert API 前, 开发者必须明确这个弹出框需要给用户多少个选择。如果只给一个或者不给选择, 它就是弹出确认框; 如果给了多个选择, 它就是弹出选择框。

3.9.1 弹出确认框

弹出确认框的实现很简单, 见代码 3-15。

代码 3-15:

```
.....
userPressConfirme: function() {
  Alert.alert(
    '弹出框标题提示语',
    '弹出框正文提示语'
    [
      {text: '我知道了', onPress: this.option1Selected }
    ]
  )
}
```

```

    });
  },
  option1Selected: function() {
    console.log('option1Selected.');
```

.....

在 iOS 平台下运行，效果如图 3-4 所示。

用户点击“我知道了”按钮后，option1Selected 函数会被执行。

我们可以在使用 Alert API 时，不提供按钮名称和回调函数，这时弹出确认框的按钮会自动为“OK”按钮。如图 3-5 所示是 Android 平台下的运行效果。



图 3-4 弹出确认框 iOS 平台运行效果

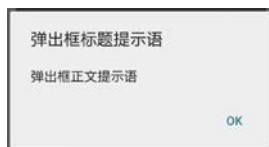


图 3-5 弹出确认框 Android 平台运行效果

3.9.2 弹出选择框

在 Android 平台下，通过 Alert API 实现的弹出选择框最多只能有三个选项；而在 iOS 平台下，弹出选择框没有选项的个数限制。

调用 Alert API 的示例见代码 3-16。

代码 3-16:

```

.....
userPressConfirme: function() {
  Alert.alert(
    '弹出框标题提示语',
    '弹出框正文提示语',
    [
      {text: '选项一', onPress: this.option1Selected },
      {text: '选项二', onPress: this.option2Selected },
      {text: '选项三', onPress: this.option3Selected },
      {text: '选项四', onPress: this.option4Selected, style: 'cancel' },
      {text: '选项五', onPress: this.option5Selected },
    ]
  );
},
option1Selected: function() {
  console.log('option1Selected.');
```

.....

```
Option4Selected: function() {  
    console.log('option4Selected.');
```

```
},  
Option5Selected: function() {  
    console.log('option5Selected.');
```

```
},.....
```

在 iOS 平台下运行，效果如图 3-6 所示。因为我们给选项四设置了 `style: 'cancel'`，所以它被排列到了最下方。

在 Android 平台下运行，效果如图 3-7 所示。注意分配给选项一的空间比较大，分配给选项一和选项二的空间比较小。而其他的选项被直接丢弃。



图 3-6 弹出选择框 iOS 平台运行效果

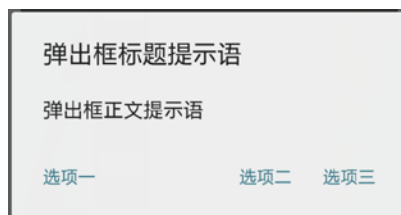


图 3-7 弹出选择框 Android 平台运行效果

在 Android 平台下运行时，将某选项的 `style` 设为 `cancel` 没有任何作用。当弹出框显示时，按手机的返回键会让弹出框消失，并且不会触发任何给 `Alert API` 提供的回调函数。

3.10 带导航栏的页面导航

敏锐的读者应当注意到，本章讨论的内容已经可以用来实现带导航栏的导航框架了。但是还有些基本知识点需要先讨论，因此这里不打算继续讨论如何实现带导航栏的导航框架。我们将在第 5 章中实现带导航栏的导航框架。

第 4 章

混合开发基础篇

我们假设在注册例程中需要执行的注册操作，已经在移动应用的上一个版本中使用原生代码开发完成了，因此没有必要将注册流程再用 React Native 开发一次。当用户输入完手机号码与密码，点击“确定”按钮后，React Native 的 RegisterLeaf 组件要能够调用原生代码提供的接口，将手机号码与密码传递给原生代码，然后等待原生代码执行注册流程。当原生代码执行完注册流程后，要能够调用 React Native 模块提供的接口传递注册结果。这就是混合开发。

在本章中，我们将看到 React Native 代码是如何做到与原生代码通信，并在 React Native 开发的界面与原生代码开发的界面之间自由切换的。我们会分别在 iOS 平台与 Android 平台实现：

- React Native 代码向原生代码发送消息，要求让用户选择联系人；
- 原生代码调用手机操作系统提供的选择联系人功能让用户选择联系人，这时应用程序界面从 React Native 开发的界面切换为原生代码开发的界面；
- 原生代码处理用户选择联系人的结果，取出联系人的姓名与电话号码，发送消息给 React Native 代码，并且释放原生代码开发的界面。此时应用程序界面回到 React Native 开发的界面；
- React Native 接收原生代码发送的消息，得到联系人的姓名与电话号码。

为了向读者展示 React Native 框架的强大开发能力，笔者将混合开发基础篇的内容放在了本章，但本章讨论的知识点与其他章节没有承前启后的关系，读者可以在需要实现混合开发时再来阅读。阅读本章要求读者具有一定的 iOS 平台 Objective-C 语言开发基础和 Android 原生应用程序开发基础。

本章讨论的内容局限于 React Native 代码与原生代码之间的桥接、通信、界面切换。更高级的混合开发知识将在第 17 章中讨论。

4.1 iOS 平台混合开发

iOS 平台开发语言分为 Objective-C 和 Swift。其中 Objective-C 是大家都很熟悉的开发语言；而 Swift 是苹果新推出的开发语言，拥有一些比较先进的语言特性。

本章讨论的 iOS 平台混合开发，因为不涉及混合开发的高级特性，所以不会讨论如何与 Swift 语言进行混合开发。原因很简单，与 Swift 语言混合开发等于与 Objective-C 语言混合开发。开发

者的 React Native 侧代码目前只能与 Objective-C 代码交互消息,在 Objective-C 侧再利用 Objective-C 语言与 Swift 语言混合开发。

因为 Swift 语言的某些先进特性,这种方式实现的 React Native、Objective-C、Swift 三种语言混合开发是完全值得的,并且不会有什么性能损失。在第 17 章中,我们会看到这种混合开发的实例。

本节的例程将实现 React Native 组件向使用 Objective-C 语言开发的模块发送消息,调用使用 Objective-C 语言开发的界面,iOS 移动应用用户在 Objective-C 界面中操作完成后,Objective-C 界面模块再被卸载,重新回到 React Native 界面,同时用户在 Objective-C 界面中的操作结果以消息的形式通知给 React Native 的指定组件。

为了能给读者更多的帮助,本例程将调用使用 Objective-C 语言开发的调用系统联系人界面模块,它在手机的联系人中选择一个联系人并选取该联系人的一个电话号码,然后将这个电话号码返回给 React Native 组件。

4.1.1 与 iOS 侧原生代码消息互通

为了实现 React Native 代码与 Objective-C 代码的消息互通,我们需要建立一个原生语言模块负责与 React Native 桥接。这个模块就是一个实现了 RCTBridgeModule 协议(Protocol)的类,其中 RCT 是 ReaCT 的缩写。

在第 3 章实现的例程基础上进入项目目录的 iOS 子目录,点击打开 Xcode 工程。在 Xcode 项目文件列表中右击,选择“New File...”,然后选择“Cocoa Touch Class”,如图 4-1 所示。

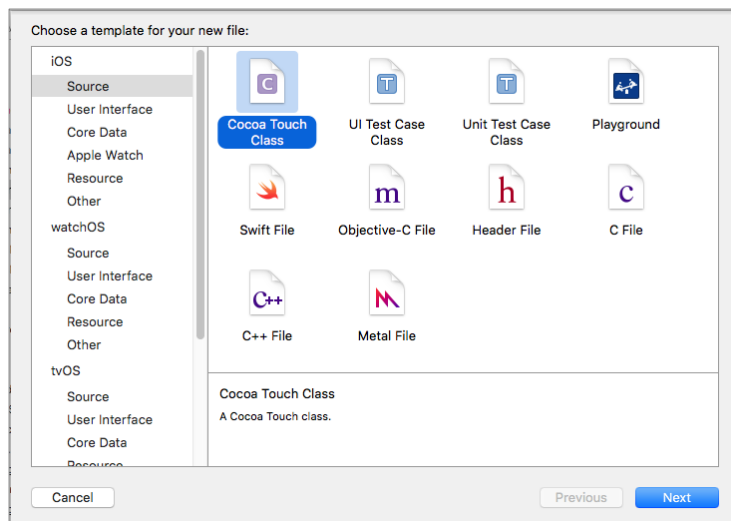


图 4-1 新建类第一步界面图

单击“Next”按钮,然后输入 Objective-C 类名 ExampleInterface,修改它的父类为 NSObject,如图 4-2 所示。

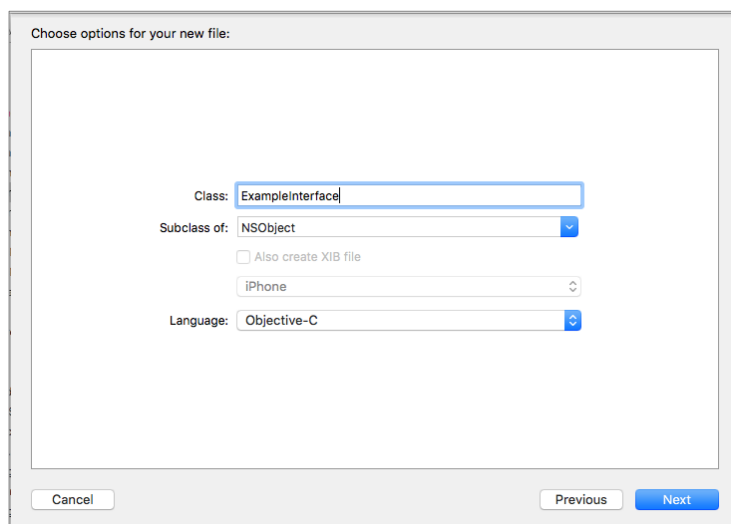


图 4-2 新建类第二步界面图

单击“Next”按钮，会出现选择保存位置对话框，如图 4-3 所示。

通常不需要改变保存位置，直接使用默认位置，单击“Create”按钮。

为了调用 iOS 操作系统提供的挑选联系人界面，我们还需要建立一个名为 CallAdressbook-ViewController 的类。重复上面的步骤来建立这个类。完成类创建后的工程文件目录如图 4-4 所示。

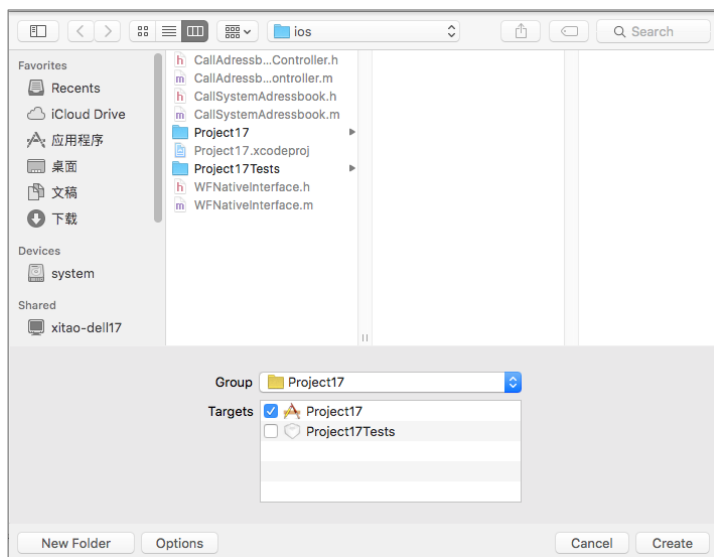


图 4-3 新建类第三步界面图

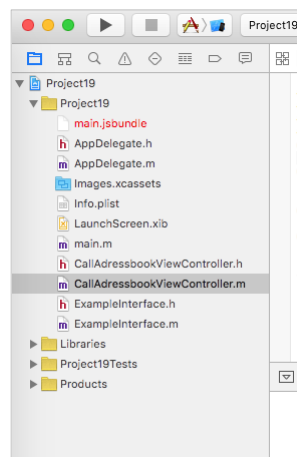


图 4-4 工程文件目录

4.1.2 React Native 代码到 iOS 原生代码的消息

修改 ExampleInterface.h，见代码 4-1。

代码 4-1, ExampleInterface.h:

```
#import "RCTBridgeModule.h" //导入这个头文件以实现 RCTBridgeModule 协议
#import "RCTBridge.h" //导入这个头文件用来实现向 React Native 发送事件
#import "RCTEventDispatcher.h" //导入这个头文件用来实现向 React Native 发送事件
@interface ExampleInterface : NSObject<RCTBridgeModule>
@property (nonatomic, strong)NSString *contactName; //用来保存所挑选的联系人姓名
@property (nonatomic, strong)NSString *contactPhoneNumber; //用来保存所挑选的联系人电话号码
@end
```

修改 ExampleInterface.m, 见代码 4-2。

代码 4-2, ExampleInterface.m:

```
#import "ExampleInterface.h"
#import "CallAddressbookViewController.h" //这是开发者使用 Objective-C 开发的类的头文件

@interface ExampleInterface ()
@property (nonatomic, strong)NSDictionary *dic;
@end

@implementation ExampleInterface
- (instancetype)init{
    return self;
}

- (NSString *)contactName{
    if (!_contactName) {
        _contactName = @" ";
    }
    return _contactName;
}

@synthesize bridge = _bridge;
//除了实现 RCTBridgeModule 协议外, 类还需要包含 RCT_EXPORT_MODULE()宏。这个宏可以添加
//一个参数用来指定在 JavaScript 中访问这个模块的名字
//通常不指定名字, 默认使用 Objective-C 类的名字
RCT_EXPORT_MODULE();
//使用 RCT_EXPORT_METHOD()宏声明需要提供给 React Native 组件调用的方法
RCT_EXPORT_METHOD(sendMessage:(NSString *)msg)
{
    //在调试窗口中打印 React Native 组件调用此函数时携带的参数
    NSLog(@"收到的来自 React Native 的消息: %@", msg);
    //检测收到的消息是否为 JSON 格式
    NSData *data = [msg dataUsingEncoding:NSUTF8StringEncoding];
    NSError *error;
    NSDictionary *dic = [NSJSONSerialization JSONObjectWithData:data options:
    NSJSONReadingMutableLeaves error:&error];
    if (error != nil) {
        NSLog(@"解析错误: %@", error);
    }
    //检测消息的 msgType 是否为 pickContact, 如果是, 则初始化挑选联系人界面
    NSString *login = [dic objectForKey:@"msgType"];
    if ([login isEqualToString:@"pickContact"]) [self callAddress];
}

- (void)callAddress{
    UIViewController *controller = (UIViewController*)[[[UIApplication sharedApplication]
    keyWindow] rootViewController];
    CallAddressbookViewController *addressbookViewController =
```

```

[[CallAddressbookViewController alloc] init];
[controller presentViewController:addressbookViewController animated:YES completion:
nil];
self.contactName = addressbookViewController.contactName;
self.contactPhoneNumber = addressbookViewController.contactPhoneNumber;
[[NSNotificationCenter defaultCenter] addObserver:self selector:@selector
(calendarEventReminderReceived:) name:@"Num" object:nil];
}
- (dispatch_queue_t)methodQueue
{
return dispatch_get_main_queue();
}
- (void)calendarEventReminderReceived:(NSNotification *)notification{
self.contactPhoneNumber = notification.object;
self.contactName = notification.userInfo[@"name"];
//去除获取到的电话号码中的特殊字符
self.contactPhoneNumber = [self.contactPhoneNumber stringByReplacingOccurrencesOfString:
@"-" withString:@""];
self.contactPhoneNumber = [self.contactPhoneNumber stringByReplacingOccurrencesOfString:
@"(" withString:@""];
self.contactPhoneNumber = [self.contactPhoneNumber stringByReplacingOccurrencesOfString:
@")" withString:@""];
self.contactPhoneNumber = [self.contactPhoneNumber stringByReplacingOccurrencesOfString:
@" " withString:@""];
NSMutableDictionary *dic = [[NSMutableDictionary alloc] init];
[dic setObject:@"pickContactResult" forKey:@"msgType"];
if (self.contactPhoneNumber == nil) {
self.contactPhoneNumber = @"";
}
[dic setObject:self.contactPhoneNumber forKey:@"peerNumber"];
if (self.contactName == nil) {
self.contactName = @"";
}
//组装发送给 React Native 侧的消息
[dic setObject:self.contactName forKey:@"displayName"];
self.dic = [dic copy];
NSError *error = [[NSError alloc] init];
NSData *data = [NSJSONSerialization dataWithJSONObject:self.dic options:0
error:&error];
NSString *str = [[NSString alloc] initWithData:data encoding:NSUTF8StringEncoding];
//向 React Native 侧发送消息
[self.bridge.eventDispatcher sendAppEventWithName:@"NativeModuleMsg" body:@{@"message":
str}];
}
@end

```

至此，我们已经实现了 Objective-C 侧的桥接工作。接下来进行 React Native 侧的桥接工作。

```

.....
//通过下面这条语句导出接口变量
var ExampleInterface = require('react-native').NativeModules.ExampleInterface;
.....
//在需要向 Objective-C 代码发送消息的地方调用下面的语句
ExampleInterface.sendMessage('{ \ "msgType\":"pickContact\ " }');
.....

```

为了使工程能够运行，还需要使用 Objective-C 语言实现的 CallAdressbookViewController.h 和 CallAdressbookViewController.m 文件。这两个文件的讲解不在本书讨论范围内，因此直接在 GitHub 中给出代码，请读者去 <https://github.com/xitaoque/4.1.2> 下载。

现在两侧的桥接都已经完成。运行代码，调试输出窗口截图如图 4-5 所示。可以看到，在图 4-5 中，React Native 代码发送给 Objective-C 代码的字符串被打印在调试窗口中，证明 Objective-C 代码收到了消息。

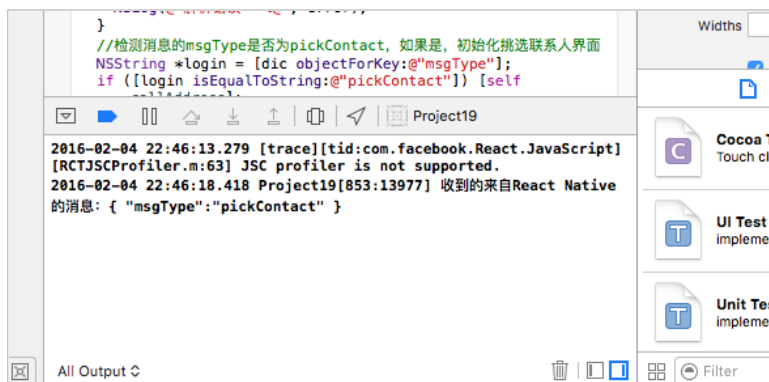


图 4-5 代码运行时 Xcode 调试输出窗口截图

图 4-5 中的调试输出表明原生代码此时已经开始运行，并且已经得到了从 React Native 侧发送的消息。这时，开发者有两个选择：

(1) 让原生代码在后台运行，直到运行结束将结果通知 React Native 侧。

(2) 原生代码初始化一个视图，并且让这个视图获得焦点，这样就实现了界面在 React Native 开发的界面与原生代码开发的界面之间切换。

4.1.3 iOS 原生代码到 React Native 代码的消息

Objective-C 代码向 React Native 代码发送消息有两种方式。其中一种方式是通过回调接口。这种方式要求 React Native 代码先将接口传递给 Objective-C 代码，然后 Objective-C 代码才可以通过这个回调接口向 React Native 代码发送消息。这使得这种方式的使用受到了限制，同时回调接口目前在 React Native 混合开发中还处于实验阶段，因此本书不讨论回调接口这种方式。

另一种方式是通过 eventDispatcher 向 React Native 模块发送事件。这能够做到 Objective-C 代码主动向 React Native 模块发送消息。

为了能够向 React Native 模块发送事件，在 Objective-C 侧，开发者需要：

```
.....  
#import "RCTBridge.h" //导入这个头文件用来实现向 React Native 发送事件  
#import "RCTEventDispatcher.h" //导入这个头文件用来实现向 React Native 发送事件  
.....  
//在业务逻辑需要的地方，向 React Native 发送事件，事件的名称由开发者指定，两侧使用同一个  
//名称就可以。本例中指定事件名称为 NativeModuleMsg
```

```
[self.bridge.eventDispatcher sendAppEventWithName:@"NativeModuleMsg" body:@{@"message":str}]];
```

在 React Native 侧，开发者需要：

```
.....
componentWillMount: function() {
  //在组件将要挂载时，导入原生代码事件接收器
  var { NativeAppEventEmitter } = require('react-native');
  //向事件接收器注册接收名为 NativeModuleMsg 的事件，并且指定收到事件后的处理函数
  this.NativeMsgSubscription = NativeAppEventEmitter.addListener(
    'NativeModuleMsg', (reminder) => {this.handleNativeInterfaceMsg(reminder.
message);}
  );
},
.....
//事件处理函数，本例只是简单地把收到的消息打印在调试日志中
handleNativeInterfaceMsg:function(aMsg) {
  console.log( 'received msg form Object-C module: '+aMsg);
},
```

如图 4-6 所示是 React Native Dev Tool 调试输出信息，也就是在 React Native 侧接收到的从原生代码发来的消息。

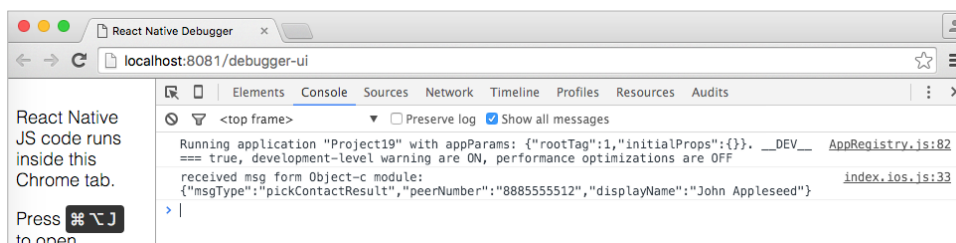


图 4-6 React Native Dev Tool 调试输出窗口截图

4.1.4 与 iOS OC 原生代码界面的切换

在前面的例程中，我们已经看到了如何实现使用不同语言开发的界面的切换。当 Objective-C 代码开始执行时，初始化一个视图并加载这个视图。这个视图会被置于 React Native 实现的视图之上，完成从 React Native 开发的界面到 Objective-C 开发的界面的切换。

当 Objective-C 开发的界面完成它的业务逻辑后，销毁这个视图，原来被覆盖的 React Native 视图再次获得焦点，完成从 Objective-C 开发的界面到 React Native 开发的界面的切换。

4.1.5 应用初始界面设定

项目初始化后，在项目文件 AppDelegate.m 中与初始化界面相关的代码示例见代码 4-3。

代码 4-3:

```
.....
RCTRootView *rootView = [[RCTRootView alloc] initWithBundleURL:JavaScriptCodeLocation
moduleNamed:@"Project19"]
```

```

        initialProperties:nil
        launchOptions:launchOptions];

self.window = [[UIWindow alloc] initWithFrame:[UIScreen mainScreen].bounds];
self.rootViewController = [UIViewController new];
self.rootViewController.view = rootView;
self.window.rootViewController = self.rootViewController;
[self.window makeKeyAndVisible];
return YES;
.....

```

代码 4-3 指定了应用初始界面是 React Native 代码的初始界面。如果有需要，开发者可以修改这段代码，让初始界面是 Objective-C 开发的界面。然后在业务逻辑需要时，创建指向 React Native 代码的视图，让这个视图获得焦点，实现界面的切换。

4.1.6 iOS 混合开发中传递的参数类型

在前面的例程中，React Native 代码与 Objective-C 代码交换的参数只有一个字符串。在实际应用中，两侧代码需要交换的参数可能会有很多个，并且有各种不同类型。

但笔者还是建议两侧代码交换的参数只是一个字符串。将这个字符串定义为 JSON 格式字符串，这样字符串参数中可以有任意类型、任意多个数据。传递前，将需要传递的参数组成一个 JSON 对象，然后通过 JSON 的转字符串方法快速转换为一个字符串，字符串传递后，在对端执行相反的操作，可以快速从字符串恢复为 JSON 对象，从而取得各参数。React Native 侧的转换、恢复方法将在 7.3.7 节、7.3.8 节中介绍。React Native 提供给 Objective-C 开发者的 RCTConvert 有一系列辅助函数，用来接收一个 JSON 值并转换为原生 Objective-C 类型或类。

如果只希望简单地传递两、三个参数，那么 RCT_EXPORT_METHOD 支持所有标准的 JSON 类型，包括：

```

string (NSString)
number (NSInteger, float, double, CGFloat, NSNumber)
boolean (BOOL, NSNumber)
array (NSArray) //包含本列表中任意类型
map (NSDictionary) //包含 string 类型的键和本列表中任意类型的值
function (RCTResponseSenderBlock)

```

Date 类型的数据不支持被直接传递，开发者需要将它转换为字符串来传递。

4.1.7 混合开发中的多线程使用

Objective-C 代码模块在被调用时不应对自己所处的线程做任何假设。React Native 的当前工作机制会使用一个单独的 GCD (Grand Central Dispatch) 串行分发队列将开发者的原生代码分发到任意线程中运行，并且将来有可能改变这种机制。如果原生代码想要指定自己在哪个线程中被调用，则可以实现 `-(dispatch_queue_t)methodQueue` 方法。例如：如果原生代码模块需要在主线程中工作，以调用一些必须在主线程中才能使用的 API，那么在代码中应当这样指定：

```

.....
- (dispatch_queue_t)methodQueue

```



```

{
    return dispatch_get_main_queue();
}
.....

```

如果一段原生代码操作需要执行很长时间，那么它应当声明一个独立的线程工作队列并在这个队列中工作。例如：`RCTAsyncLocalStorage` 模块在工作时会创建一个属于自己的队列，这样 `React Native` 本身的工作队列不会因为 `RCTAsyncLocalStorage` 模块进行缓慢的磁盘操作而堵塞。例如：

```

.....
- (dispatch_queue_t)methodQueue
{
    return dispatch_queue_create("com.facebook.React.AsyncLocalStorageQueue",
    DISPATCH_QUEUE_SERIAL);
}
.....

```

在原生代码中创建的工作队列会被原生代码模块中所有的方法共享。如果在原生代码模块中只有一个函数是耗时较长的（或者由于某种原因必须不同的工作队列中运行），开发者可以在函数体内通过 `dispatch_async` 方法来指定它在另一个工作队列中执行。这种指定不影响原生代码模块中的其他方法。例如：

```

.....
RCT_EXPORT_METHOD(doSomethingExpensive:(NSString *)param callback:
(RCTResponseSenderBlock)callback)
{
    dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^ {
        .....//耗时很长的操作
        [self.bridge.eventDispatcher sendAppEventWithName:.....
        //可以在任何时候向 React Native 模块发送处理结果
    });
}
.....

```

提示：跨模块共同使用同一个分发队列

原生代码模块中的 `methodQueue` 方法只会在模块被初始化时执行一次，初始化得到的引用会被 `React Native` 的桥梁机制保存。开发者通常不需要保存的创建的队列的引用，除非模块中还需要使用到这个队列。然而，如果开发者希望所创建的分发队列能被跨模块使用，那么就必须确保代码中保存了分发队列实例，并且为每一个需要共同使用分发队列的模块都返回了相同的分发队列实例，而且只是简单地返回一个具有相同名字的队列，无法做到跨模块共同使用同一个分发队列。

4.1.8 原生代码实现 Promise 机制

原生代码能够实现 `Promise` 机制。当我们使用 `ES 2016` 的 `async/await` 语法时，`Promise` 机制可以简化代码。

当被桥接的原生代码函数的最后两个参数是 `RCTPromiseResolveBlock` 和 `RCTPromiseRejectBlock` 时，与它配合的 `JavaScript` 的方法将返回一个 `JavaScript` 的 `Promise` 对象。

实现 Promise 机制的代码示例见代码 4-4。

代码 4-4:

```
.....
RCT_REMAP_METHOD(handleMessage:(NSString *)rnmessage,
    resolver:(RCTPromiseResolveBlock)resolve
    rejecter:(RCTPromiseRejectBlock)reject)
{
    .....//处理 rnmessage
    if (成功) {
        resolve(成功时返回的参数);
    } else {
        reject(失败时返回的携带错误信息的对象);
    }
}
.....
```

相应的，在 React Native 侧使用 Promise 机制的代码示例见代码 4-5。

代码 4-5:

```
.....
var ExampleInterface = require('react-native').NativeModules.ExampleInterface;
.....
//在向 Objective-C 代码发送消息的地方调用下面的语句
ExampleInterface.showContactView('aMessage').then( (成功时返回的参数)=> {
    .....//处理得到的参数
}
).catch((失败时返回的携带错误信息的对象)=>{
    .....//处理错误
});
.....
```

4.1.9 跨语言常量

混合开发中的原生模块可以导出一些常量，将某些在 Objective-C 原生代码中定义的常量暴露给 React Native 侧。这种方法对混合开发有一定的用处，特别是在 Objective-C 侧代码中某些常量需要与 React Native 侧代码中的对应常量保持一致时。

为了导出常量，需要在 Objective-C 侧原生代码中定义 constantsToExport 函数。示例如下：

```
- (NSDictionary *)constantsToExport
{
    return @{@"constantName": @"constantValue" };
}
```

在 React Native 侧，可以通过 moduleName.constantName 来得到 Objective-C 侧定义的常量。开发者需要注意：常量的导出只发生在应用初始化时，在应用中，如果在 Objective-C 侧改变了 constantsToExport 函数返回的值，对 React Native 侧常量的值不会有任何影响。

在 iOS 平台混合开发中，开发者还可以把通过 Objective-C 语言定义的枚举类型导出到 React Native 侧。但是鉴于 Android 平台混合开发中没有这种机制，建议开发者在 iOS 平台混合开发中

不使用这种技术，只导出普通的常量。

在 iOS 平台混合开发中导出常量的例程可参见 17.1.2 节中的代码 17-2。

4.2 Android 平台混合开发

现在我们来讨论如何实现 Android 平台的混合开发。

首先打开 Android Studio，点击“Open an existing Android Studio project”，如图 4-7 所示。

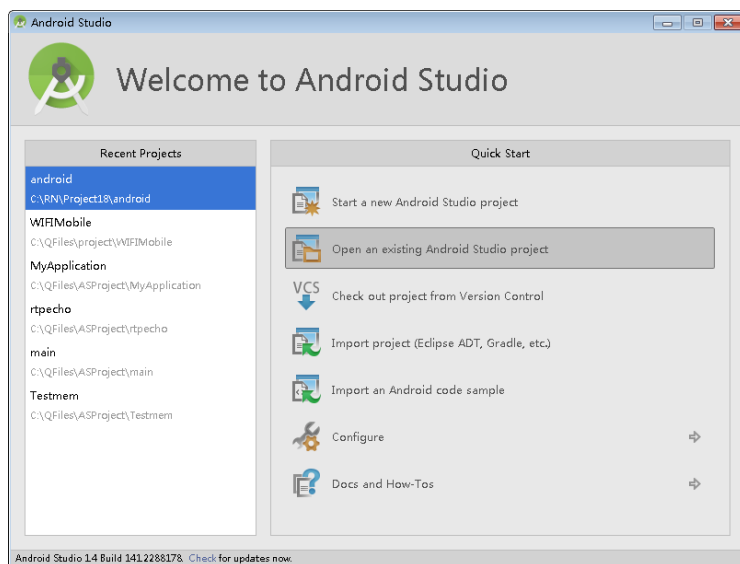


图 4-7 打开 React Native 项目 Android 工程

然后在 React Native 项目目录下选择 android 子目录下的 build.gradle 文件，单击“OK”按钮，如图 4-8 所示。

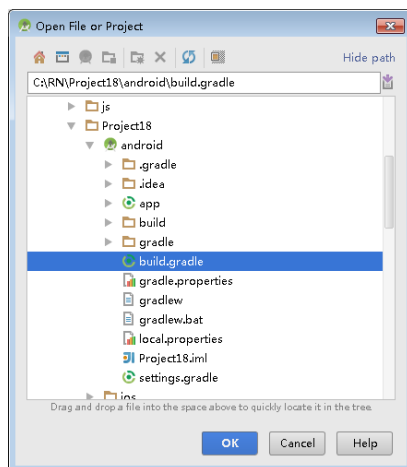


图 4-8 选择工程文件

React Native 项目对应的 Android 工程就已经被打开了，打开后的界面如图 4-9 所示。

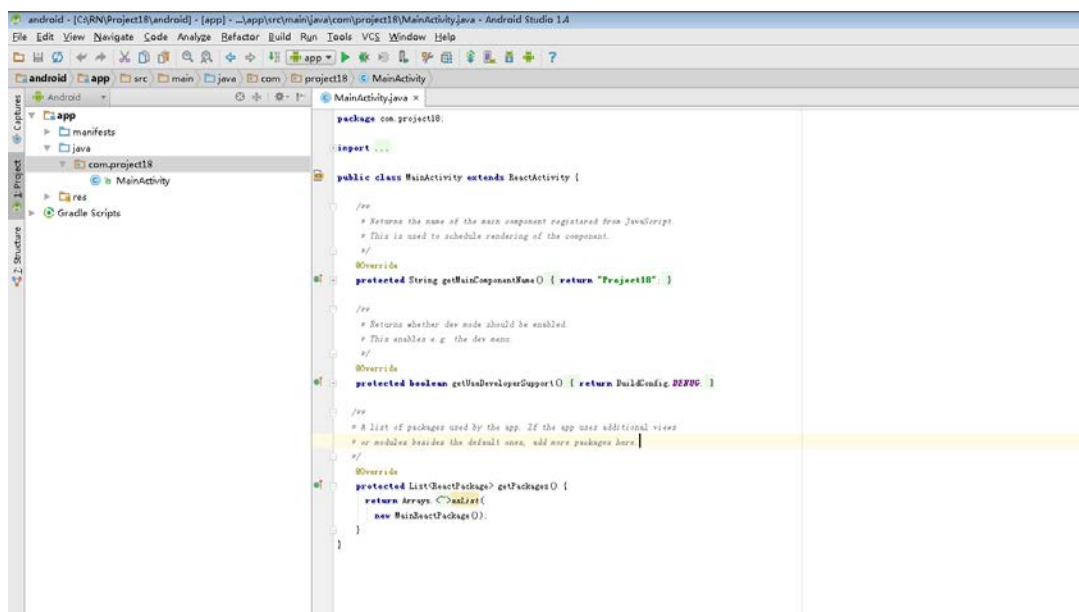


图 4-9 打开工程后应当出现的界面

4.2.1 与 Android 原生代码消息互通

为了实现 React Native 侧代码与 Android 侧代码互通，我们需要在 Android 侧建立接口类。

首先创建一个新的类，如图 4-10 所示。

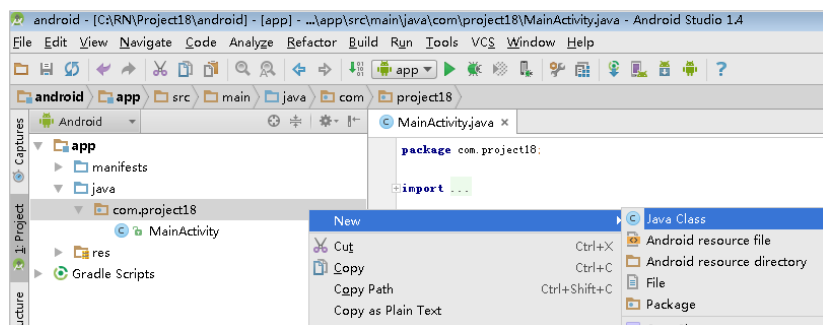


图 4-10 创建一个新类步骤一

打开新建类对话框，在输入框中输入类名，然后单击“OK”按钮，如图 4-11 所示。

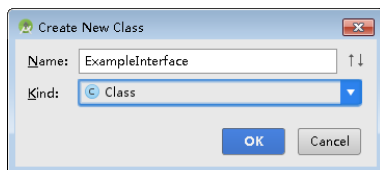


图 4-11 创建一个新类步骤二

创建好类后，界面如图 4-12 所示。

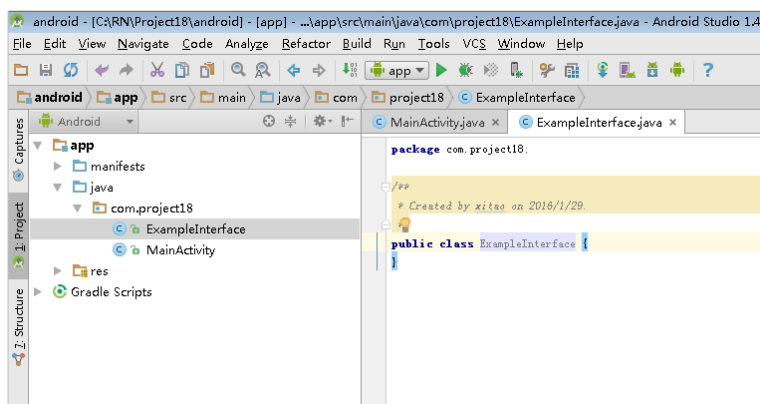


图 4-12 创建新类后的界面

4.2.2 React Native 代码到 Android 原生代码的消息

Android 侧的接口类需要继承 `ReactContextBaseJavaModule` 类，然后实现 React Native 侧希望调用的方法。示例见代码 4-6。

代码 4-6:

```
package com.Project19;
import android.util.Log;
import com.facebook.react.bridge.NativeModule;
import com.facebook.react.bridge.ReactApplicationContext;
import com.facebook.react.bridge.ReactContext;
import com.facebook.react.bridge.ReactContextBaseJavaModule;
import com.facebook.react.bridge.ReactMethod;
public class ExampleInterface extends ReactContextBaseJavaModule {
    public ExampleInterface(ReactApplicationContext reactContext) {
        super(reactContext);
    }
    @Override
    public String getName() {
        return "ExampleInterface";
    }
    @ReactMethod
    public void HandleMessage(String aMessage ) { //这里是原生代码处理消息的函数
        Log.i("RNMessage", "received message from RN:"+aMessage );//本例中只是简单地在
        //日志中把收到的消息打印出来
    }
}
```

继承 `ReactContextBaseJavaModule` 的类都需要实现 `getName` 函数，这个函数的用途是返回原生代码模块的名称，在 React Native 侧使用这个名称来调用原生代码模块提供的其他函数。

Android 侧接口类中的函数必须要使用“`@ReactMethod`”关键字将它注释为一个 React 函数才能在 React Native 侧被调用。并且这些函数不能有返回值，因为被调用的原生代码是异步执行的，所以原生代码执行结束时只能通过回调函数或者发送消息将执行结果通知给 React Native 侧。

Android 侧实现混合开发的最后一步是注册开发者编写的原生代码模块。为此开发者需要实现一个 React 包管理类，这个类继承自 `ReactPackage`，并且必须实现 `createNativeModules` 方法。实际的注册过程将在 `createNativeModules` 函数被调用时发生。任何在 React Native 侧希望被调用的原生代码模块都需要进行注册。注册代码示例见代码 4-7。

代码 4-7:

```
package com.Project19;
import com.facebook.react.ReactPackage;
import com.facebook.react.bridge.jsModule;
import com.facebook.react.bridge.NativeModule;
import com.facebook.react.bridge.ReactApplicationContext;
import com.facebook.react.uimanager.ViewManager;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
public class AnExampleReactPackage implements ReactPackage {
    @Override
    public List<NativeModule> createNativeModules(ReactApplicationContext reactContext)
    {
        List<NativeModule> modules = new ArrayList<>();
        modules.add(new ExampleInterface(reactContext)); //在这里加入开发的接口
        return modules;
    }
    @Override
    public List<Class<? extends JavaScriptModule>> createJSModules() {
        return Collections.emptyList();
    }
    @Override
    public List<ViewManager> createViewManagers(ReactApplicationContext reactContext)
    {
        return Collections.emptyList();
    }
}
```

开发者编写的 React 包管理器的实例需要在 `MainActivity.java` 的 `getPackages` 函数中被创建。创建示例见代码 4-8。

代码 4-8:

```
.....
/**
 * A list of packages used by the app. If the app uses additional views
 * or modules besides the default ones, add more packages here.
 */
@Override
protected List<ReactPackage> getPackages() {
    return Arrays.<ReactPackage>asList(
        new MainReactPackage(),
        new AnExampleReactPackage()); //开发者需要添加这一行代码
    }
.....
```

在 React Native 侧，开发者可以如代码 4-9 所示调用 Android 侧实现的原生代码。

代码 4-9:

```
var { NativeModules } = require('react-native');
let ExampleInterface = NativeModules.ExampleInterface;

buttonPressed:function() {
  NativeModules.ExampleInterface.HandleMessage( 'testMessage');
}
```

完整的 React Native 侧代码见代码 4-10。

代码 4-10:

```
'use strict';
var React = require('react-native');
var { NativeModules } = require('react-native');
let ExampleInterface = NativeModules.ExampleInterface;
var {
  AppRegistry, StyleSheet, Text, View,
} = React;
var Project19 = React.createClass({
  render: function() {
    return (
      <View style={styles.container}>
        <Text style={styles.welcome}
          onPress={this.buttonPressed}>
          Welcome to React Native!
        </Text>

        </View>
      );
    },
  buttonPressed:function() {
    NativeModules.ExampleInterface.HandleMessage( 'testMessage');
  }
});

var styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#F5FCFF',
  },
  welcome: {
    fontSize: 30,
    textAlign: 'center',
    margin: 10,
    backgroundColor:'grey'
  }
});
AppRegistry.registerComponent('Project19', () => Project19);
```

运行应用后，在手机屏幕上点击，可以在 Android Studio 的日志中看到从 React Native 侧发来的消息，见图 4-13 中倒数第 2 行。

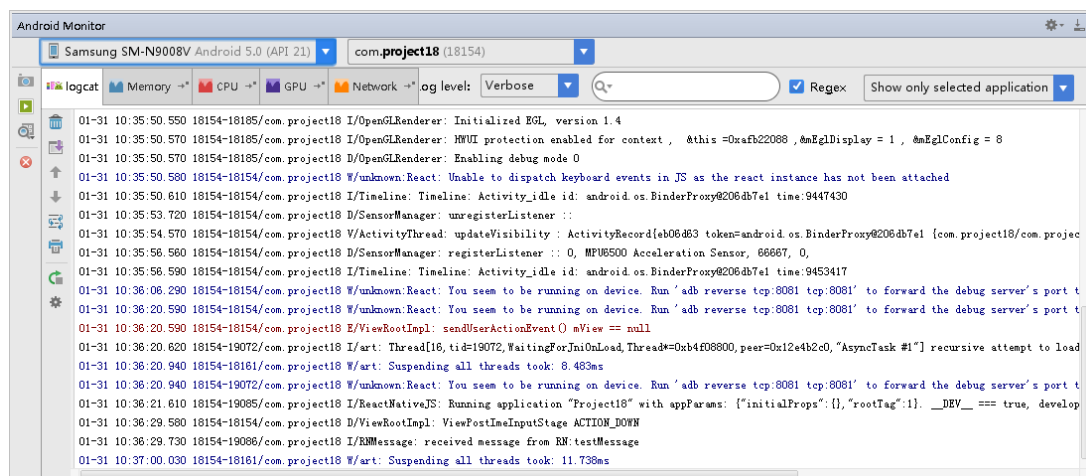


图 4-13 原生代码日志输出

图 4-13 中的调试输出表明原生代码此时已经开始运行，并且已经得到了从 React Native 侧发送的消息。这时，开发者有两个选择：

(1) 让原生代码在后台运行，直到运行结束将结果通知 React Native 侧。使用这种方法，React Native 实现的界面仍然在手机屏幕上呈现并能接受用户的输入。

(2) 让原生代码初始化一个视图，并且让这个视图获得焦点，这样就实现了应用界面在 React Native 开发的界面与原生代码开发的界面之间切换。

注意：在 Android Studio 中编译项目时，会结束用户在命令行窗口运行的服务器。因此开发者需要在 Android Studio 编译完成后，再次在命令行窗口通过“react-native start”命令启动服务器。

4.2.3 与 Android 原生代码界面的切换

接下来我们讨论如何在混合开发中实现界面的切换。

首先在项目中新建一个 Activity。选择项目包名并右击，选择“New”→“Activity”→“Empty Activity”，如图 4-14 所示。

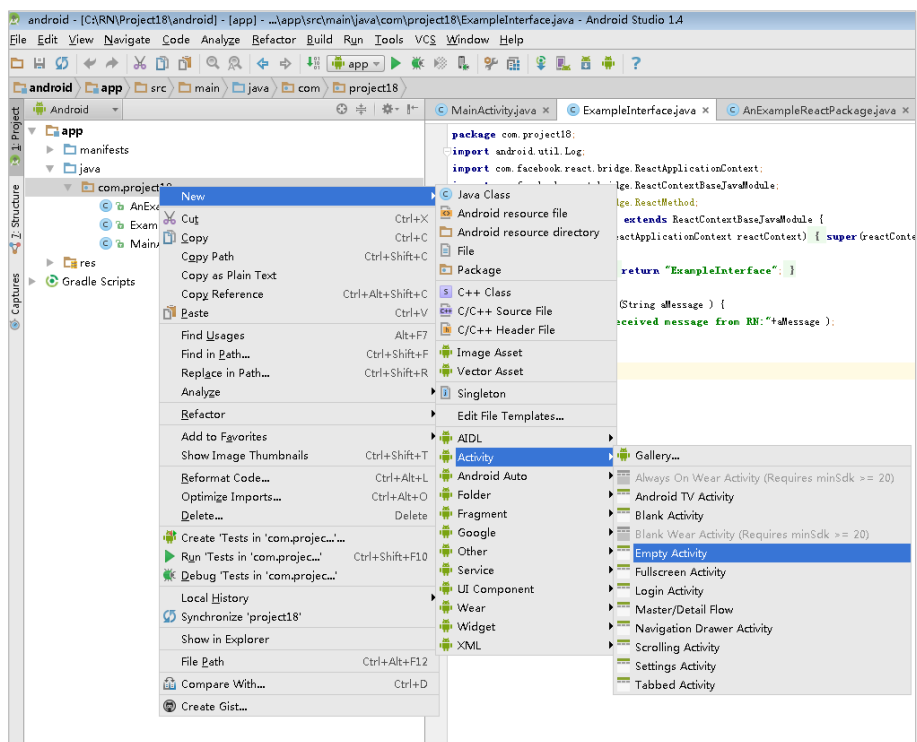


图 4-14 创建新的 Activity

设定新建 Activity 的界面如图 4-15 所示。

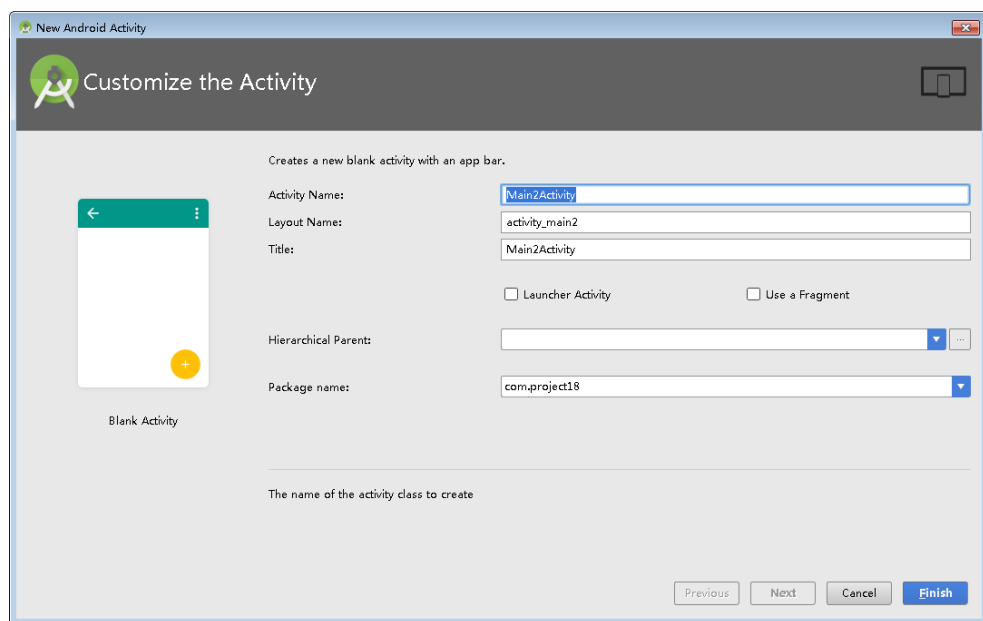


图 4-15 设定新建 Activity 的界面

不进行任何修改，使用 Android Studio 提供的值，然后单击“Finish”按钮。

新的 Activity 建立后，Android 工程项目文件列表如图 4-16 所示。

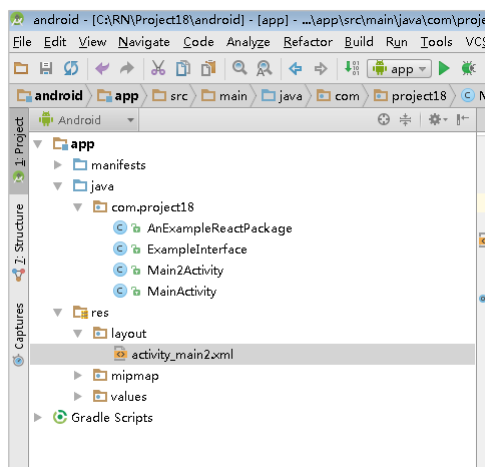


图 4-16 Android 工程项目文件列表

打开 Main2Activity 文件，修改代码见代码 4-11。

代码 4-11:

```
package com.Project19;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
public class Main2Activity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main2);
    }
    public void Btn1_Click(View view) {
        this.finish();    //结束本 Activity
    }
}
```

再打开 activity_main2.xml，将一个 Button 控件拖入手机屏幕中央，如图 4-17 所示。

双击这个按钮，修改它的显示字符串。然后选中这个 Button，在 Android Studio 的右下侧列表中找到 onClick 事件，点击将它与 Btn1_click 函数挂接，如图 4-18 所示。

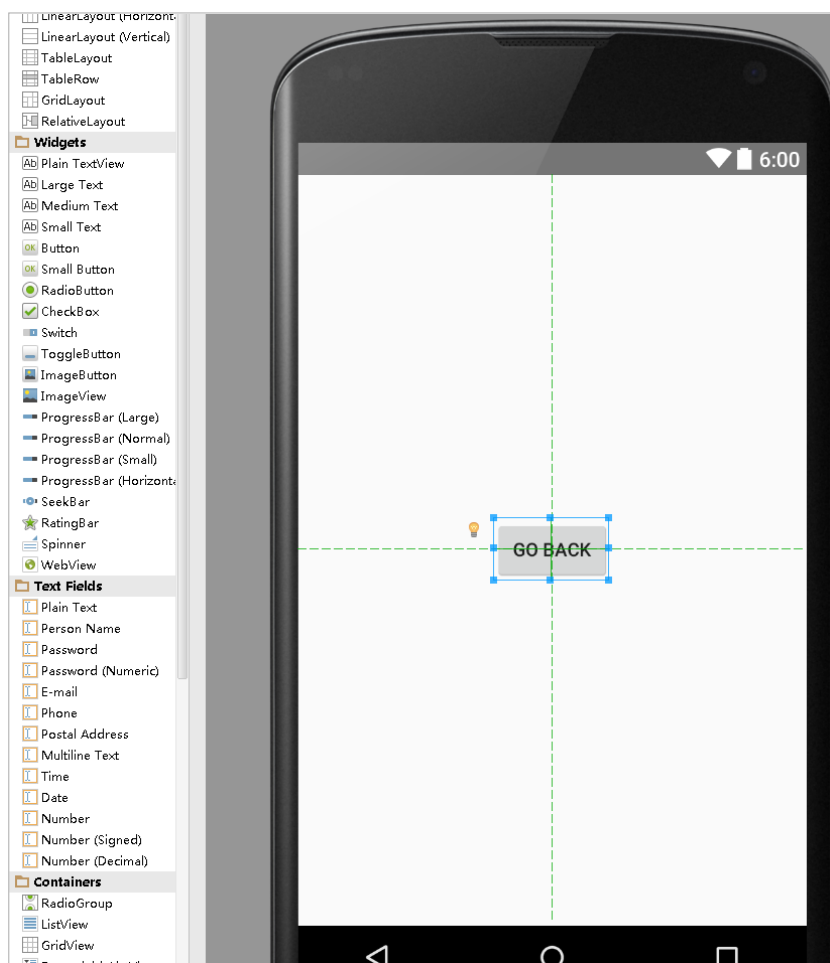


图 4-17 创建一个 Button

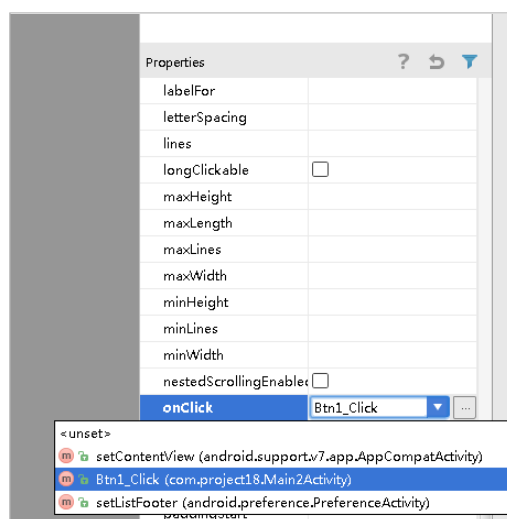


图 4-18 挂接点击事件

最后修改 ExampleInterface.java 文件，见代码 4-12。

代码 4-12:

```
package com.Project19;
import android.content.ComponentName;
import android.content.Intent;
import android.util.Log;
import com.facebook.react.bridge.ReactApplicationContext;
import com.facebook.react.bridge.ReactContextBaseJavaModule;
import com.facebook.react.bridge.ReactMethod;
public class ExampleInterface extends ReactContextBaseJavaModule {
    ReactApplicationContext aContext;
    public ExampleInterface(ReactApplicationContext reactContext) {
        super(reactContext);
        aContext = reactContext;    //保存 MainActivity 传来的上下文实例
    }
    @Override
    public String getName() {
        return "ExampleInterface";
    }
    @ReactMethod
    public void HandleMessage(String aMessage ) {
        Log.i("RNMessage", "received message from RN:" + aMessage);
        Intent aIntent = new Intent( aContext, Main2Activity.class);
        aIntent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
        aContext.startActivity( aIntent );    //切换到 Main2Activity
    }
}
```

修改完成后，编译并运行应用程序，点击两个不同界面上的按钮，就可以实现 React Native 开发的界面与原生代码开发的界面的切换。

4.2.4 Android 原生代码到 React Native 代码的消息

现在，同样在 Android 平台下实现使用原生代码调用系统选择联系人界面，并且在用户选择联系人后，将用户选择的联系人的姓名与电话号码传递到 React Native 侧。

首先需要给 Android 工程加入读写联系人权限（注意，虽然只是读，但是仍然需要有写联系人权限，否则在某些三星机型上会报错）。修改 React Native 项目目录下的\android\app\src\main\AndroidManifest.xml 文件的头部，见代码 4-13。

代码 4-13:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.Project19" >
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.READ_CONTACTS" /> //增加读取联系人
    权限
<application
    .....
```

修改 ExampleInterface.java，见代码 4-14。

代码 4-14:

```
package com.Project19;
import android.content.Intent;
import android.os.Bundle;
import android.provider.ContactsContract;
import android.util.Log;
import com.facebook.react.bridge.ReactApplicationContext;
import com.facebook.react.bridge.ReactContextBaseJavaModule;
import com.facebook.react.bridge.ReactMethod;
import com.facebook.react.modules.core.DeviceEventManagerModule;
public class ExampleInterface extends ReactContextBaseJavaModule {
    ReactApplicationContext aContext;
    public ExampleInterface(ReactApplicationContext reactContext) {
        super(reactContext);
        aContext = reactContext;
    }
    @Override
    public String getName() {
        return "ExampleInterface";
    }
    @ReactMethod
    public void HandleMessage(String aMessage ) {
        Log.i("RNMessage", "received message from RN:" + aMessage);
        Intent aIntent = new Intent(Intent.ACTION_PICK);
        aIntent.setType(ContactsContract.Contacts.CONTENT_TYPE);
        Bundle b=new Bundle();           //这个 Bundle 没有用，但必须要有
        aContext.startActivityForResult(aIntent, 1, b);    //调用系统提供的选择联系人界面
    }
    public void sendMessage( String aMessage ) { //此函数用来向 React Native 侧发送消息
        aContext.getJSModule(DeviceEventManagerModule.RCTDeviceEventEmitter.class)
            .emit("AndroidToRNMessage", aMessage);
    }
}
```

当用户点击 React Native 界面上的按钮后，会启动系统提供的选择联系人界面。这个界面是系统提供的，也是手机用户平时经常用、非常熟悉的界面。

选择联系人界面被启动后，我们需要接收用户选择联系人的操作结果。因为是从 MainActivity 跳转到选择联系人界面的，所以选择的结果会被发往 MainActivity。开发者接收选择结果的函数需要在 MainActivity.java 中实现。修改 MainActivity.java，见代码 4-15。

代码 4-15:

```
package com.Project19;
import android.content.Intent;
import android.database.Cursor;
import android.net.Uri;
import android.provider.ContactsContract;
import android.util.Log;
import com.facebook.react.ReactActivity;
import com.facebook.react.ReactPackage;
import com.facebook.react.shell.MainReactPackage;
```

```

import java.util.Arrays;
import java.util.List;
public class MainActivity extends ReactActivity {
    AnExampleReactPackage anExampleRNPackage;    //新增一个成员变量

    @Override
    protected String getMainComponentName() {
        return "Project19";
    }

    @Override
    protected boolean getUseDeveloperSupport() {
        return BuildConfig.DEBUG;
    }

    @Override
    protected List<ReactPackage> getPackages() {
        anExampleRNPackage = new AnExampleReactPackage();    //新增成员变量赋值
        return Arrays.<ReactPackage>asList(
            new MainReactPackage(),
            anExampleRNPackage);    //新增成员变量进入队列
    }

    @Override    //这个函数取得用户的选择结果，然后传递给接口以向 React Native 侧发送消息
    public void onActivityResult(int requestCode, int resultCode, Intent data) {
        super.onActivityResult(requestCode, resultCode, data);
        //下一条语句，如果不是选择联系人的结果，或者选择联系人未成功，则直接退出
        if ( requestCode != 1 || resultCode != RESULT_OK) return;
        Uri contactData = data.getData();
        Cursor cursor = managedQuery(contactData, null, null, null, null);
        cursor.moveToFirst();
        String toRNMessage = this.getContactInfo(cursor);    //取得结果字符串
        anExampleRNPackage.rnExampleInterface.sendMessage( toRNMessage); //交接口发送
    }

    // getContactInfo 函数取出用户选择的联系人的姓名与电话号码，按某规则生成结果字符串
    private String getContactInfo(Cursor cursor) {
        String name = "";
        String phoneNumber = "";
        int idColumn = cursor.getColumnIndex(ContactsContract.Contacts._ID);
        String contactId = cursor.getString(idColumn);
        String q = ContactsContract.CommonDataKinds.Phone.CONTACT_ID + "=" + contactId;
        Uri aUri = ContactsContract.CommonDataKinds.Phone.CONTENT_URI;
        Cursor phone = getContentResolver().query(aUri, null, queryString, null, null);
        String dn= ContactsContract.Contacts.DISPLAY_NAME;
        String pn = ContactsContract.CommonDataKinds.Phone.NUMBER;
        if (phone.moveToFirst()) {
            for (; !phone.isAfterLast(); phone.moveToNext()) {
                dn =
                    name = cursor.getString(cursor.getColumnIndex(dn));
                phoneNumber = phone.getString(phone.getColumnIndex(pn));
            }
            phone.close();
        }
        String result = "{ \"msgType\": \"pickContactResult\", \"displayName\": \"" +
name + "\", \"phoneNumber\": \"" + phoneNumber + "\" }";
        return result;
    }
}

```

提示：开发者可以通过 4.2.8 节中讨论的方法在 ExampleInterface.java 中监听启动的 Activity 返回的结果。两种方法各有优缺点，由开发者自己灵活掌握。

我们还需要对 AnExampleReactPackage.java 进行修改以实现 MainActivity 调用接口向 React Native 侧发送消息。修改后的代码见代码 4-16。

代码 4-16:

```
package com.Project19;
import com.facebook.react.ReactPackage;
import com.facebook.react.bridge.jsModule;
import com.facebook.react.bridge.NativeModule;
import com.facebook.react.bridge.ReactApplicationContext;
import com.facebook.react.uimanager.ViewManager;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
public class AnExampleReactPackage implements ReactPackage {
    public ExampleInterface rnExampleInterface;                //新增成员变量
    @Override
    public List<NativeModule> createNativeModules(ReactApplicationContext reactContext) {
        List<NativeModule> modules = new ArrayList<>();
        rnExampleInterface = new ExampleInterface(reactContext);    //新增成员变量赋值
        modules.add(rnExampleInterface);                            //新增成员变量加入列表中
        return modules;
    }
    @Override
    public List<Class<? extends JavaScriptModule>> createJSModules() {
        return Collections.emptyList();
    }
    @Override
    public List<ViewManager> createViewManagers(ReactApplicationContext reactContext) {
        return Collections.emptyList();
    }
}
```

在 React Native 侧，接收原生代码消息的代码见代码 4-17。

代码 4-17:

```
.....
componentWillMount:function() {
    console.log('componentWillMount');
    DeviceEventEmitter.addListener('AndroidToRNMessage',this.handleAndroidMessage);
},
handleAndroidMessage:function(aMessage) {
    console.log('handleAndroidMessage:'+aMessage);
},
.....
```

注意：监听消息类型必须与代码 4-14 倒数第三行中发送时指定的消息类型完全一致。

在这里，我们只是简单地把接收到的消息打印出来。打印出来的消息格式如下：

```
{ "msgType":"pickContactResult", "displayName":"10086", "phoneNumber":"10086" }
```

这是一个 JSON 格式的消息。本书 7.3.8 节中将讨论如何快速利用它生成一个 JSON 对象，以方便开发者处理。

4.2.5 应用初始界面设定

Android 应用初始界面在 React Native 项目目录下的 `\android\app\src\main\AndroidManifest.xml` 文件中进行设置。文件内容中含有“`intent-filter`”字段所在的 Activity 就是默认启动的第一个 Activity。

通过配置这个字段，开发者可以设定应用初始界面是 React Native 开发的界面还是原生代码开发的界面。

4.2.6 传递的参数类型

与 4.1.6 节中描述的理由相同，使用一个 JSON 字符串传递各种参数比传递多个参数要更便于快速开发。如果开发者需要，下面的参数类型在 `@ReactMethod` 注明的方法中会被直接映射到对应的 JavaScript 类型。

```
Boolean -> Bool
Integer -> Number
Double -> Number
Float -> Number
String -> String
Callback -> function
ReadableMap -> Object
ReadableArray -> Array
```

4.2.7 回调函数与 Promise 机制

Android 平台混合开发支持从 Android 侧调用 React Native 侧提供的回调函数。在讨论回调函数的实现机制前，先介绍一下回调函数的缺点。

当 React Native 侧向原生代码模块提供了回调函数后，原生代码模块应当只调用这个回调函数一次。可以先将这个回调函数保存起来，在需要时再调用。但只应当调用一次，多次调用会有不可预期的结果。

回调函数需要先由 React Native 侧提供，然后 Android 侧才能调用这个回调函数。也就是说，Android 侧无法主动通过回调函数向 React Native 侧发送消息。这也是我们推荐使用消息机制而不是回调函数的主要原因。

在 React Native 混合开发中，即使原生代码正确地调用了回调函数，在 React Native 侧对应的回调函数也不会立即执行。因为混合开发中的桥接机制是异步的。

如果开发者还是希望使用回调函数，在 Android 侧它的使用方法请参考代码 4-18。

代码 4-18:

```
.....
public class SomethingManagerModule extends ReactContextBaseJavaModule {
```



```

.....
@ReactMethod
public void aMethodForRN(String aPara, Callback aCallback) {
    .....//按业务逻辑处理
    aCallback.invoke( msg);    //调用回调函数，返回结果
}
.....

```

在 React Native 侧，回调函数的使用方法请参考代码 4-19。

代码 4-19:

```

.....
SomethingManager.aMethodForRN( aPara,
    (msg) => {
        console.log(msg);
    });
.....

```

在 Android 平台混合开发中，同样支持使用原生代码实现 Promise 机制。当被桥接的原生代码函数的最后一个参数类型为 Promise 时，这个函数会返回一个 JavaScript 的 Promise 对象给与它对应的 JavaScript 方法。

使用 Promise 机制的原生代码请参考代码 4-20。

代码 4-20:

```

.....
public class SomethingManagerModule extends ReactContextBaseJavaModule {
    .....
    @ReactMethod
    public void aMethodForRN(String aPara, Promise promise) {
        try{
            .....//按业务逻辑处理，当 React Native 侧希望的操作失败时，抛出异常
            promise.resolve(msg);    //对应 React Native 侧的 Promise 执行成功
        } catch (Exception e) {
            .....//处理异常，生成相应的 error，记录失败信息
            promise.reject(error);    //对应 React Native 侧的 Promise 执行失败
        }
    }
}
.....

```

相应的，在 React Native 侧使用 Promise 机制的代码见代码 4-21。

代码 4-21:

```

.....
SomethingManager.aMethodForRN( aPara).then((msg)=>
    {
        .....//处理得到的参数
    }
    ).catch((error)=>{
        .....//处理错误
    });
.....

```

4.2.8 监听 ActivityResult 与 Android 生命周期事件

在代码 4-14 中，我们启动了操作系统提供的选择联系人的 Activity，并且在 MainActivity 中监听选择的结果。开发者也可以选择直接在 Android 与 React Native 的接口类中监听启动 Activity 的结果。为此，ExampleInterface.java 需要实现 ActivityEventListener 接口，并且调用相应方法。代码示例见代码 4-22。

代码 4-22:

```
.....
public class ExampleInterface extends ReactContextBaseJavaModule implements
ActivityEventListener {
    public ExampleInterface(ReactApplicationContext reactContext) {
        super(reactContext);
        reactContext.addActivityEventListener(this);
    }
    @Override
    //注意三个参数都是 final
    public void onActivityResult(final int requestCode, final int resultCode, final Intent
intent) {
        // 不需要 super.onActivityResult(requestCode, resultCode, data);
        .....//按业务逻辑处理
    }
}
```

在 Android 平台 React Native 开发中，开发者还可以在 Android 与 React Native 的接口类中监听本应用的生命周期事件。为此，ExampleInterface.java 需要实现 ActivityEventListener 接口，然后实现三个生命周期事件处理接口函数就可以满足这个功能需求。

监听生命周期事件的代码示例见代码 4-23。

代码 4-23:

```
.....
public class ExampleInterface extends ReactContextBaseJavaModule implements
ActivityEventListener {
    componentWillMount:function() {
        reactContext.addLifecycleEventListener(this);
    },
    @Override
    public void onHostResume() {
        .....// 应用 onResume 时的处理
    },
    @Override
    public void onHostPause() {
        .....//应用 onPause 时的处理
    },
    @Override
    public void onHostDestroy() {
        .....//应用 onHostDestroy 时的处理
    }
    .....
}
```

4.2.9 混合开发中的多线程机制

在进行 Android 平台混合开发时，原生代码模块不应当假设自己会在什么线程中执行。如果原生代码模块执行需要较长的时间，则会占用比较多的 CPU 处理资源，原生代码模块应当自己启动一个线程并在该线程中执行。

4.2.10 跨语言常量

在 Android 平台下的混合开发中，可以通过实现名为 `getConstants` 的成员函数，将某些在 Android 原生代码中定义的常量暴露给 React Native 侧。这种方法对混合开发有一定的用处，特别是在 Android 侧和 React Native 侧需要保持某些常量一致时。

在 Android 侧代码实现类似于：

```
@Override
public Map<String, Object> getConstants() {
    final Map<String, Object> constants = new HashMap<>();
    constants.put("constantName", constantValue);
    return constants;
}
```

而在 React Native 侧，开发者可以通过 `nativeModuleName.constantName` 来访问被导出的常量。

第 5 章

flexbox 布局、View、Image 与可触摸组件

React Native 利用 flexbox 模型布局,也就是在手机屏幕上对组件进行排列。利用 flexbox 模型,开发者可以开发出动态宽高自适应的 UI 布局。

View 组件是 React Native 组件中最基础的组件。后面讨论到其他组件时,读者会发现很多组件都支持 View 组件的属性。

Image 组件用来在 UI 上显示图像,它的重要性不言而喻。

可触摸组件用来把其他组件变成对用户触摸有反应的组件。

本章将详细讨论 flexbox 模型与这些组件。

5.1 flexbox 布局

我们在前面已经介绍过 View、Text、TextInput 组件的一些属性设置了。React Native 强大的 UI 布局能力主要是通过各个不同组件的样式 (style) 属性中各个键的设置来实现的。而大部分组件的 style 都支持 flexbox 布局样式设置。

样式设置都是 JSON 格式的数据,也就是一个个“键: 值”对。在后续章节中都用键和值来称呼它们。当我们说某个键是某类型时,指的是那个键对应的值是某类型。

flexbox(弹性盒)是 W3C 提出的 UI 设计模型规范的一种实现,有布局神器的美誉。React Native 实现了其中的大部分功能。下面先统一讨论一下大部分组件都支持的样式设置。

代码 5-1 供读者用来在其中加入我们接下来将要讨论的各种样式键与对应的值,然后在手机上运行,实时查看效果。有需要时,可以加入更多的子 View。

代码 5-1:

```
'use strict';
var React = require('react-native');
var {
  AppRegistry, View, StyleSheet,
} = React;
```

```

var Project19 = React.createClass({
  render: function() {
    return (
      <View style={styles.container}>           //定义根 View
        <View style={styles.test1} />          //定义子 View
        <View style={styles.test2} />          //定义子 View
      </View>
    );
  },
});
var styles = StyleSheet.create({
  container: {                                //根 View 样式定义
    flex: 1,
    backgroundColor: 'green',
  },
  test1: {                                    //子 View 样式定义
    width: 360,
    height: 60,
    backgroundColor: 'red',
  },
  test2: {                                    //子 View 样式定义
    width: 40,
    height: 40,
    backgroundColor: 'blue',
  },
});
AppRegistry.registerComponent('Project19', () => Project19);

```

5.1.1 位置及宽、高相关样式键

对于位置，首先要讨论的是 `position` 键。它是字符串类型，可以取值为 `relative`（默认值）或者 `absolute`，表示当前描述的位置是相对定位还是绝对定位的。

与位置相关的样式设置键还有：`top`、`bottom`、`left`、`right`。它们都是数值类型，表示描述的位置从左（或者右）多少位置显示，还有从上（或者下）多少位置显示。

当 `position` 键的值为 `absolute` 时，描述位置可以使用 `top`、`bottom`、`left`、`right` 四个键，表示当前组件的位置距父组件的最上（下、左、右）沿有多少 `pt`。

当 `position` 键的值为 `relative` 时，不可以使用 `bottom` 和 `right` 键继续描述位置。`Top` 和 `left` 键的默认值为 0。`Top` 和 `left` 键表示当前组件距离上一个同级组件的最上（左）沿有多少 `pt`。

与宽、高相关的数值类型键有：`width`、`height`、`maxHeight`、`maxWidth`、`minHeight`、`minWidth`。因为使用 `flexbox` 布局，组件的宽和高是可以动态改变的，所以可以设置宽和高的最大值与最小值。

在 `flexbox` 布局中，允许存在宽、高动态改变的组件，所以有些组件可以不指定宽、高和最大/最小宽、高，这时这些组件的宽、高由它们内部需要显示的内容动态决定。在 `flexbox` 布局中，要尽量不指定组件的宽、高，而由组件的内容及其排列方式来动态决定组件的宽、高。如果这一点做不到，那么就尽可能只指定宽、高中的一个值，另一个值动态决定。

5.1.2 决定子组件排列规则的键

在组件中通过对某些键的赋值可以决定其子组件的排列方式。

通过 flexbox 布局来排列子组件的优点在于，不需要指定子组件的绝对位置，也不需要根据运行时屏幕大小动态计算子组件的大小；而是通过排列一个个长方形（正方形是一种特殊的长方形）来进行布局。

在 UI 设计中，开发者经常需要处理宽、高动态变化的组件。因为宽、高是动态变化的，所以开发者在设计时不能知道该组件的宽、高是多少，只能让它的布局及其同级组件的布局都动态变化。这时，将这些组件排列在一个 flexbox 布局的父组件中，能很好地适应这种情况，让各个组件都能正确、美观地展示出来。

5.1.2.1 flexDirection 键

flexDirection 键决定了组件内部的子组件是如何排列的，它的取值可以为 row、column，对应于图 5-1 中左起的第一个图和第三个图。而 W3C 提出的 row-reverse 和 column-reverse 则不支持。如果在 View 的样式里没有定义 flexDirection，则取默认值 column。



图 5-1 flexDirection 键工作示意图

如图 5-1 所示，对于父组件来说，它的子组件就像是一个个长方形（或者正方形）的盒子，父组件的 flexDirection 决定了它们如何排列。我们在 UI 中看到的圆形或者其他形状的组件，都是长方形的组件通过图片遮盖或者透明露出部分底色之类的技巧实现的。

5.1.2.2 flexWrap 键

在默认情况下，组件中的子组件按照 flexDirection 键决定的方向一直排列下去，即使超出了该方向的宽度或者高度也不管。

对 flexWrap 键赋值可以改变这种情况。它可以取两个字符串类型值中的一个：wrap 或 nowrap。默认值是 nowrap，表示不会自动换行（或者换列）排列。flexDirection 为 row、flexWrap 为 wrap 时的工作示意图如图 5-2 所示。

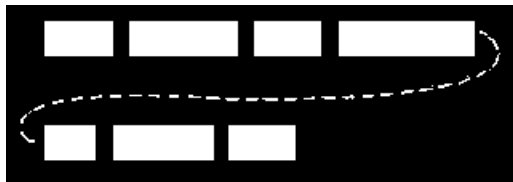


图 5-2 flexWrap 键工作示意图

当 flexWrap 取值为 wrap 时需要注意一点，当父组件右侧（flexDirection 为 row）或者父组件

下侧（`flexDirection` 为 `column`）的所剩空间不足以显示下一个组件时，下一个组件会跳到下一行或者下一列头显示，从而导致在原来的行或者列中留下一部分空白。这个问题可以通过设置背景色、背景图片来解决。

`flexWrap` 取值为 `wrap` 时，还需要与下面即将介绍的 `alignItem` 键配合才能实现子组件自动换行排列，并且 `alignItem` 键不能取值为“`stretch`”。

5.1.2.3 justifyContent 键

`justifyContent` 键用来定义在一个方向上如何排列子组件。它有 5 种可能的字符串类型值：`flex-start`、`flex-end`、`center`、`space-between`、`space-around`，它们对应的布局示意图如图 5-3 所示。其中，黑色块表示一个父组件的范围，白色块表示子组件。

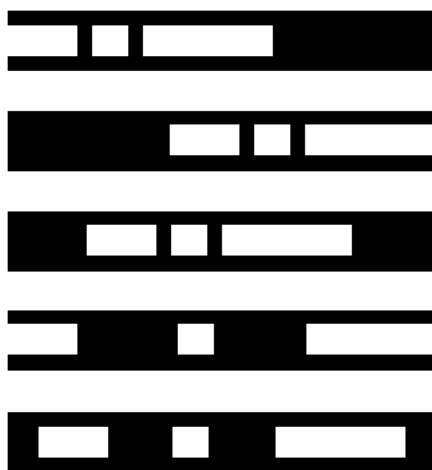


图 5-3 justifyContent 键工作示意图

5.1.2.4 alignItems 键

`alignItems` 键定义了 View 组件中所有子组件的对齐规则。它有 4 种可能的字符串类型值：`flex-start`、`flex-end`、`center`、`stretch`。其中，`flex-start` 代表顶部对齐；`flex-end` 代表底部对齐；`center` 代表中部对齐；`stretch` 代表拉长对齐。它们的布局示意图如图 5-4 所示。

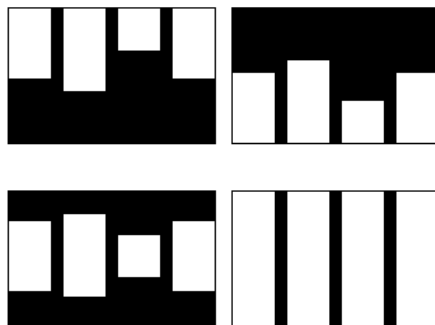


图 5-4 alignItems 键工作示意图

5.1.3 决定组件显示规则的键

在 5.1.2 节中讨论的样式键都是决定组件中子组件显示规则的键。下面再介绍两个用来决定组件显示规则的样式键。

5.1.3.1 “灵活的” flex 键

flex 键的类型是数值，取值为 0 或者 1，默认值是 0。当它的值为 1 时，子组件将自动缩放以适应父组件剩下的空白空间。

开发者使用 flex 键时一定要小心，它的自动缩放意味着不仅可以改变自己的宽、高与位置，还可以改变与它同级的其他组件的位置。

开发者想在手机屏幕上均匀地分布高度为 50 的灰色长条，应当怎么做？

这很简单，使用 flexbox 布局，见代码 5-2。

代码 5-2:

```
.....
let Project19 = React.createClass({
  render: function() {
    return (
      <View style={styles.container}>          //在根 View 中放置 5 个子 View
        <View style={styles.vs}/>
        <View style={styles.vs}/>
        <View style={styles.vs}/>
        <View style={styles.vs}/>
        <View style={styles.vs}/>
      </View>
    );
  }
});
var styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'space-around',
    backgroundColor: 'white',
  },
  vs: {
    height: 50,
    backgroundColor: 'gray'
  },
  vs1: {                                //这种样式在后面会用到
    flex: 1,
    backgroundColor: 'black'
  }
});
.....
```

我们在 5 个子 View 的父组件中声明了 `justifyContent: 'space-around'`，5 个子 View 都使用同一种样式 `vs`，只设置了高度与背景颜色。这段代码的运行结果如图 5-5 所示。

现在应用需要第四个子 View 能占满第三个子 View 与第五个子 View 中的所有空白，并且第 4 个子 View 的显示颜色为黑色。开发者可以对第四个子 View 不设置高度，设置 flex:1，再设置颜色，于是第四个子 View 的描述被修改成<View style={styles.vs1}/>。然后运行代码，结果如图 5-6 所示。

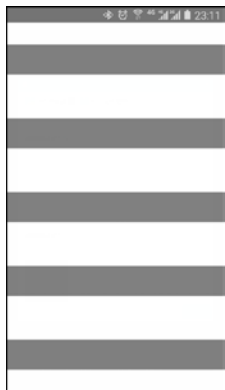


图 5-5 代码 5-2 运行结果

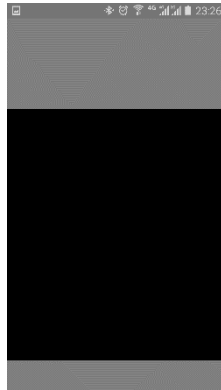


图 5-6 修改后代码 5-2 运行结果

在修改后的代码中，样式 vs1 中的 flex:1 不仅让第四个子 View 沿着它的父组件规定的 flexDirection 方向扩展到相邻同级组件，并且它还顶着其相邻同级组件继续扩展，直到把所有的同级组件都顶结实了才收手。在图 5-6 中，所有的空白空间都没有了，或者说都被第四个子 View 吞下了。

开发者需要注意的是，设置了 flex:1 的组件并不是只能扩展，在后面可以看到，设定为固定宽、高的组件也可以在应用执行中动态改变宽、高。如果固定宽、高的组件在执行中宽、高增加了，那么设置了 flex:1 的组件就会缩小来适应其同级组件的宽、高变化。

5.1.3.2 alignSelf 键

alignSelf 键有 5 种可能的字符串类型值：auto、flex-start、flex-end、center、stretch。其用途是让组件忽略它的父组件样式中 alignItems 键的取值，而对该组件使用 alignSelf 键对应的规则。当它取值为 auto 时，表示使用父 View 组件的 alignItems 值，其他 4 个值的含义与 alignItems 相同。图 5-7 显示了当父组件的 alignItems 设置为 flex-start，而第三个子组件的 alignSelf 设置为 flex-end 时的情况。



图 5-7 alignSelf 键工作示意图

5.1.4 边框、空隙与填充

我们可以指定一个组件的边框（border）、空隙（margin）和填充（padding）的各种属性（如果支持的话）。

空隙是组件与组件之间的空间，而填充是内容与边框之间的空间，请参见图 5-8。

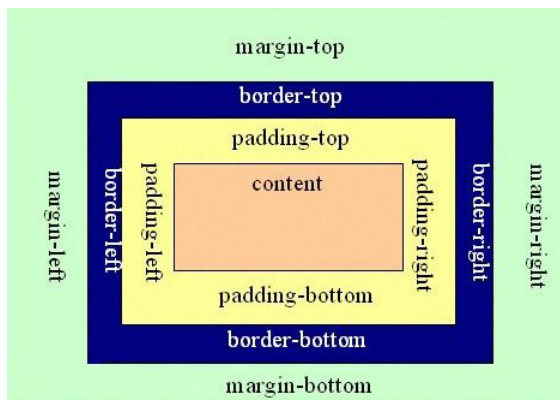


图 5-8 边框、空隙、填充位置示意图

因为边框、空隙与填充样式设置简单，并且在后面的应用例程中经常会用到，所以本节只是介绍它们的特性，不提供例程了。

5.1.4.1 边框宽度

定义边框宽度可以使用到 5 个数值类型的键，它们是 `borderWidth`、`borderTopWidth`、`borderRightWidth`、`borderBottomWidth` 和 `borderLeftWidth`。

5.1.4.2 填充宽度

定义元素边框与元素内容之间的空间可以使用 7 个数值类型的键。其中，`padding` 对四个边都有效；`paddingHorizontal` 对水平方向两个边有效；`paddingVertical` 对垂直方向两个边有效；`paddingBottom`、`paddingLeft`、`paddingRight` 和 `paddingTop` 各控制一个边的填充。

5.1.4.3 空隙宽度

定义组件与组件之间的空隙可以使用 7 个数值类型的键。7 个值的作用与 `padding` 类似，这里不再详述，只列出它们的键名：`margin`、`marginHorizontal`、`marginVertical`、`marginBottom`、`marginLeft`、`marginRight`、`marginTop`。

5.1.5 组件多样式声明与动态样式声明

在前面的例程中，每个组件的样式都大致是：

```
.....
style={styles.container}>
.....
var styles = StyleSheet.create({
  container: {
.....
```

这种写法的优点是简洁明了。对于样式一直保持不变的组件，推荐使用这种用法。

组件可以有多种样式，写法如下：

```
style=[styles.container, styles.aStyle, styles.bStyle]}
```

这样声明的多种样式，React Native 将会对多种样式做一个合并的操作。在多种样式中，如果对某一个键有重复定义，那么后面的样式定义的键将覆盖前面的。

在 UI 设计中，有些组件的样式在应用运行中是需要改变的，这时我们需要直接在描述组件的 JSX 代码中写入需要变化的样式。见代码 5-3。

代码 5-3:

```
.....
<Text
  onPress={this.onTextPress}
  style=[styles.aStyle, {color: this.state.appColor1}]}> //color 样式键可以动态变化
.....
onTextPress:function() {
  this.setState({
    appColor1: 'green'
  })
},
.....
```

5.2 View 组件

View 组件是 React Native 最基本的组件。绝大部分其他 React Native 组件都继承了 View 组件的属性，包括支持 View 组件的样式设置、回调函数及其他属性。

学习 View 组件的使用不是一件容易的事情。View 组件是其他组件的基础，在学习其他组件前需要先了解 View 组件；但 View 组件的很多特性又从其他组件中反映出来的，这就要求读者可能需要不止一次地阅读本章中的内容。第一次学习时如果有些概念不清楚，则可以先暂时跳过继续向下学习，当学习到其他组件使用从 View 组件继承而来的属性时，可以再回过头来看这些属性。

5.2.1 View 组件的颜色与边框

backgroundColor 键用来指定组件的背景颜色。如果没有指定，默认的背景颜色会是一种非常浅的灰色。从 React Native 0.19.0 开始，只有 Text 与 TextInput 组件会继承其父组件的背景颜色，这意味着其他组件都需要设置自己的背景颜色。

Opacity 键定义了 View 组件的透明度，它的取值为 0~1。当值为 0 时，表示组件完全透明；而值为 1 时，表示组件完全不透明。一个组件透明表示被它遮盖住的组件可以透过它显示出来。

borderStyle 键用来设置边框的风格，它只能取 solid、dotted 和 dashed 三个值之一，分别表示实线边框、点状边框和虚线边框。这个键的默认值为 solid。

定义边框颜色可以使用到 5 个字符串类型的键，它们是 borderColor（通常只使用这个）、borderTopColor、borderRightColor、borderBottomColor 和 borderLeftColor。

定义圆角边框可以使用到 5 个数值类型的键，它们是 `borderRadius`（通常只设置这个）、`borderTopLeftRadius`、`borderTopRightRadius`、`borderBottomLeftRadius` 和 `borderBottomRightRadius`。

定义圆角时有一个比较有用的用法：如果一个 View 的宽、高相等，值为 $2X$ ，将 `borderRadius` 的值设为 X 时，这个 View 在显示上会是一个圆。这种效果不仅可以在 View 上实现，也可以在下面要讨论的 Image 组件上实现。

代码 5-4 示范了上面讨论的各个样式键值的效果。

代码 5-4，`index.ios.js` 或者 `index.android.js`：

```
'use strict';
var React = require('react-native');
var {
  AppRegistry, StyleSheet, Text, View,
} = React;
var Project19 = React.createClass({
  render: function() {
    return (
      <View style={styles.container}> //根 View 中的每个子 View 透明度递增
        <View style={styles.welcome}
          opacity={0}/> //透明度值为 0 的 View 组件不可见
        <View style={styles.welcome}
          opacity={0.1}/>
        <View style={styles.welcome}
          opacity={0.25}/>
        <View style={styles.welcome}
          opacity={0.5}/>
        <View style={styles.welcome}
          opacity={0.75}/>
        <View style={styles.welcome}
          opacity={1}/>
        <View style={styles.welcome}
          opacity={5}/>
      </View>
    );
  }
});
var styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'space-around', //注意这个样式键值的显示效果
    alignItems: 'center',
    backgroundColor: 'grey',
  },
  welcome: {
    width: 50,
    height: 50,
    borderWidth: 1,
    backgroundColor: 'white',
    borderRadius: 25
  }
});
AppRegistry.registerComponent('Project19', () => Project19);
```

代码 5-4 运行效果如图 5-9 所示。

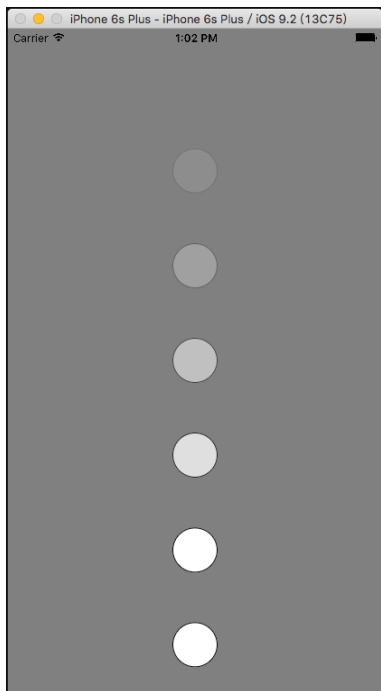


图 5-9 代码 5-4 运行效果

注意，在图 5-9 中，从下到上，随着透明度值的减小，子 View 遮盖住的父 View 的灰色越来越多地透视出来，直到透明度值为 0 的子 View 组件完全不可见自己的颜色。读者可以观察到，透明度不仅影响组件的显示，还影响组件的边框显示。

因为绝大部分组件都继承了 View 组件的属性，所以 View 组件的样式设定对绝大部分组件也是有同样效果的。

补充：如何让一个组件的边框只有一个实际像素的宽度呢？答案很简单，设置 `borderWidth:1/pixelRatio`。注意，`pixelRatio` 是通过 `PixelRatio.get()` 得到的 pt 值。

5.2.2 View 组件的阴影与其他视觉效果

`shadowColor`、`shadowOffset`、`shadowOpacity`、`shadowRadius` 是与阴影相关的样式键，分别对应着组件的阴影颜色、阴影位移值、阴影透明度与阴影圆角率。在 UI 设计中让一个 View 具有阴影效果没有什么意义。在 View 组件中定义这些样式键的意义更多的在于让继承它的其他组件去各自实现这些效果。Text 组件就继承了这些样式键，我们将在 6.1.4 节中看到组件的阴影效果。

`overflow` 键有两个字符串类型的取值：`visible` 和 `hidden`。它定义了当 View 组件中的子组件宽超出 View 组件宽高时的行为。默认是 `hidden`，即隐藏超出部分。这个键只对 iOS 平台有效，在 Android 平台上即使被设为 `visible`，显示的特性也仍然是 `hidden`。

在图 5-10 中，上图是一个 Text 组件的 `overflow` 样式键为 `hidden` 时的效果，可以看到有两个

字母的显示有缺失；下图是同一个 Text 组件的 overflow 样式键为 visible 时的效果，可以看到原来缺失的部分显示出来，并且显示在组件范围之外（超出部分背景颜色为白色）。

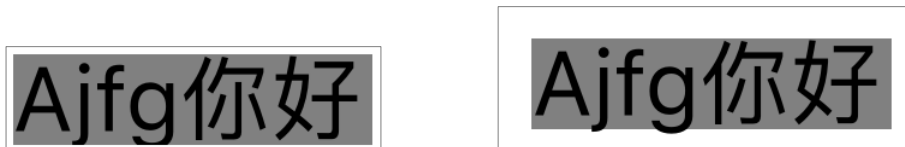


图 5-10 overflow 键的效果

backfaceVisibility 键有两个字符串类型的取值：visible 和 hidden。它的设置与 UI 动画效果有关。本章暂时不讨论这个键的使用。

elevation 是 Android 平台特有的样式键。它是数值类型的样式键，通过在组件周围渲染阴影让用户产生一种组件浮现在手机屏幕上的视觉效果。在图 5-11 中，两个黑色块的样式定义分别是：

```
.....
block1: {
  width:150,
  height:150,
  borderWidth:1,
  backgroundColor:'black',
  borderRadius:25
},
block2: {
  width:150,
  height:150,
  borderWidth:1,
  backgroundColor:'black',
  borderRadius:25,
  elevation: 50      //两样式唯一区别，block2 有 elevation 键值
}
.....
```

唯一的区别是下面的黑色块设置了 elevation 值，可以看到该黑色块周围有阴影效果。

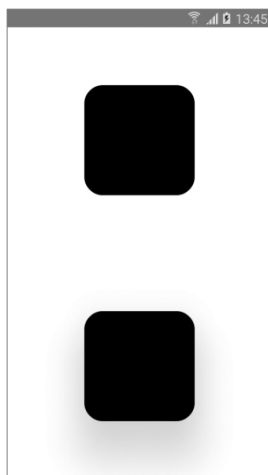


图 5-11 elevation 键的效果

5.2.3 View 组件的变形

在 React Native 开发中，开发者可以利用 transform 样式键设置实现组件的变形，从而实现文字或图像的变形。变形包括：translate（平移）、scale（缩放）、rotate（旋转）、skew（倾斜）四种类型。

transform 样式键设置的格式是：

```
transform:
[
  {perspective: number},
  {rotate: string},
  {rotateX: string},
  {rotateY: string},
  {rotateZ: string},
  {scale: number},
  {scaleX: number},
  {scaleY: number},
  {translateX: number},
  {translateY: number},
  {skewX: string},
  {skewY: string}
]
```

使用 View 组件来显示变形的效果不会很理想，但绝大部分组件都继承了 View 组件的属性。代码 5-5 使用 Text 组件来演示变形的效果。

代码 5-5，index.ios.js 或者 index.android.js:

```
'use strict';
var React = require('react-native');
var {
  AppRegistry, StyleSheet, Text, View,
} = React;
var Project19 = React.createClass({
  render: function() {
    return (
      <View style={styles.container}>
        <Text style={styles.welcome0}>
          Welcome to React Native!
        </Text>
        <Text style={styles.welcome1}>
          Welcome to React Native!
        </Text>
        <Text style={styles.welcome2}>
          Welcome to React Native!
        </Text>
        <Text style={styles.welcome3}>
          Welcome to React Native!
        </Text>
        <Text style={styles.welcome4}>
          Welcome to React Native!
        </Text>
        <Text style={styles.welcome5}>
          Welcome to React Native!
        </Text>
      </View>
    );
  }
});
```

```

        </Text>
        <Text style={styles.welcome6}>
            Welcome to React Native!
        </Text>
        <Text style={styles.welcome7}>
            Welcome to React Native!
        </Text>
        <Text style={styles.welcome8}>
            Welcome to React Native!
        </Text>
        <Text style={styles.welcome9}>
            Welcome to React Native!
        </Text>
        <Text style={styles.welcome10}>
            Welcome to React Native!
        </Text>
    </View>
    );
}
});

var styles = StyleSheet.create({
    container: {
        flex: 1, justifyContent: 'center', alignItems: 'center',
        backgroundColor: '#F5FCFF',
    },
    welcome0: {
        flex: 1, justifyContent: 'center', alignItems: 'center',
        transform: [{rotate:'45deg'}] //不指定轴旋转
    },
    welcome1: {
        flex: 1, justifyContent: 'center', alignItems: 'center',
        transform: [{rotateX:'45deg'}] //X 轴旋转
    },
    welcome2: {
        flex: 1, justifyContent: 'center', alignItems: 'center',
        transform: [{rotateY:'45deg'}] //Y 轴旋转
    },
    welcome3: {
        flex: 1, justifyContent: 'center', alignItems: 'center',
        transform: [{rotateZ:'45deg'}] //Z 轴旋转
    },
    welcome4: {
        flex: 1, justifyContent: 'center', alignItems: 'center',
        transform: [{scale:2}] //X,Y 轴都放大
    },
    welcome5: {
        flex: 1, justifyContent: 'center', alignItems: 'center',
        transform: [{scaleX:2}] //X 轴放大
    },
    welcome6: {
        flex: 1, justifyContent: 'center', alignItems: 'center',
        transform: [{scaleY:2}] //Y 轴放大
    },
    welcome7: {

```



```

        flex: 1, justifyContent: 'center', alignItems: 'center',
        transform: [{translateX:200}] //X 轴平移
    },
    welcome8: {
        flex: 1, justifyContent: 'center', alignItems: 'center',
        transform: [{translateY:150}] //Y 轴平移
    },
    welcome9: {
        flex: 1, justifyContent: 'center', alignItems: 'center',
        transform: [{skewX:'45deg'}] //X 轴倾斜
    },
    welcome10: {
        flex: 1, justifyContent: 'center', alignItems: 'center',
        transform: [{skewY:'45deg'}] //Y 轴倾斜
    }
  }
});
AppRegistry.registerComponent('Project19', () => Project19);

```

代码 5-5 运行效果参见图 5-12。



图 5-12 代码 5-5 运行效果截图

transform 键的 perspective 元素与 3D 变形效果有关，在例程中暂不涉及。

在代码 5-5 中，角度用“Xdeg”来表示，X 取值都是 45，开发者可以用 0~360 中的任何值取代。特别值得一提的是，如果设定 transform: [{rotateY: '90deg'}]，就可以让字符串或者图像改为沿垂直方向显示。

在代码 5-5 中，每个组件只演示了一种变形效果。在实际开发中，可以多种变形效果叠加。例如：

```
transform: [{scale:2},{skewY:'45deg'}] // X,Y 轴都放大并且 Y 轴倾斜
```

最后讨论旋转与倾斜的区别。rotate 控制目标整体旋转，与目标内部形状无关，目标内部不发

生任何形变；而 skew 目标内部的形状会随倾斜而改变。读者可以修改代码，只留两个字符串，并适当加大字体，然后仔细观察两者的不同。

5.2.4 View 组件的回调函数

这是本书中第一次正式提到“组件的回调函数”这个概念。

假如 A 组件有一个属性，名称是 onB（在 React Native 开发中，这样命名的意思是 B 事件发生了），它要求开发者提供的 onB 的值必须是一个函数的引用，当 B 事件发生时，React Native 框架通过 onB 的值来回调这个函数。为了在称呼上简洁明了，我们直接将提供的函数称为 A 组件的 onB 回调函数，而 onB 被称为回调函数类型的属性。

onMoveShouldSetResponder、onMoveShouldSetResponderCapture、onResponderGrant、onResponderMove、onResponderReject、onResponderRelease、onResponderTerminate、onResponderTerminationRequest、onStartShouldSetResponder、onStartShouldSetResponderCapture 这 10 个属性都是回调函数类型的属性，它们被用来处理用户手势（手指在屏幕上触摸、移动）事件。但对于大部分处理用户手势事件的开发需求，开发者都是使用其他组件的能力来完成的。第 11 章将要详细讨论的用户手势识别可以被认为是在这些函数上封装了一层，让开发者能更方便地处理手势事件。这 10 个属性值几乎不会在开发中用到，因此本书不做讨论。

onLayout 是 View 组件的回调函数类型的属性。当 View 组件被加载或者布局改变时，回调函数将被调用（将在 5.2.6 节详细讨论这个函数）。

View 组件还有三个没有在官方文档中公布的回调函数：onPressStart、onPressMove、onPressEnd。没有在官方文档中公布的原因不详，有可能是 View 组件支持这三个回调函数，但继承它的其他组件很多都不支持。

开发者使用这三个函数需要冒一定的风险，因为没有在官方文档中公布，也许下一次 React Native 发布新版本时，这三个函数就没有了。但这三个回调函数能满足开发者的某些功能需求，并且 React Native 更新了这么多次版本也没有取消它们，所以使用它们的风险也不高。从这三个函数的名称就可以知道它们对应：开始触摸事件、触摸点移动事件和触摸结束事件。这三个回调函数都会收到一个 event 对象参数。在 event 对象的结构中对普通开发者有用的结构是：

```
{
  timeStamp: aNumber,           //以整数表示的时间戳，从 1970 年 1 月 1 日至时间戳的毫秒数
  nativeEvent: {
    locationX: aNumber,
    locationY: aNumber,
  }
}
```

通过传入的 event 参数，开发者可以知道事件的发生时间，以及事件发生时用户的手指在屏幕上的位置。代码 5-6 示范了这三个回调函数的使用。

代码 5-6，index.ios.js 或者 index.android.js:

```
'use strict';
var React = require('react-native');
```

```

var {
  AppRegistry, StyleSheet, Text, View, StatusBarIOS
} = React;
var Project19 = React.createClass({
  _onTouchMove:function(event) {
    console.log("touch move: " + event.timeStamp+ ' , X:'
      +event.nativeEvent.locationX+', Y:' +event.nativeEvent.locationY);
  },
  _onTouchStart:function(event) {
    console.log("touch start: " + event.timeStamp+ ' , X:'
      +event.nativeEvent.locationX+', Y:' +event.nativeEvent.locationY);
  },
  _onTouchEnd:function(event) {
    console.log("touch end: " + event.timeStamp+ ' , X:'
      +event.nativeEvent.locationX+', Y:' +event.nativeEvent.locationY);
  },
  render: function() {
    return (
      <View style={styles.container}
        onTouchStart={this._onTouchStart}
        onTouchMove={this._onTouchMove}
        onTouchEnd={this._onTouchEnd}>
      </View>
    );
  }
});
var styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#F5FCFF',
  }
});
AppRegistry.registerComponent('Project19', () => Project19);

```

代码 5-6 运行后，手指随便做个触摸屏幕、移动再松开的动作，在调试工具中日志输出参见图 5-13。

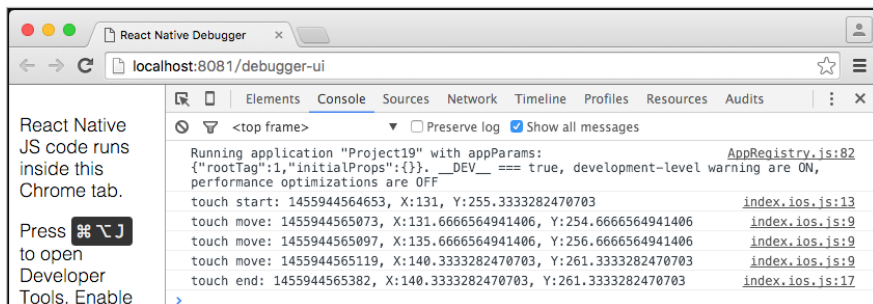


图 5-13 代码 5-6 运行日志输出

这三个回调函数都有一个特殊的特性，就是在 React Native 开发中，通常是在最上层的组件中（render 函数靠后渲染的组件）处理触摸事件的；但这三个回调函数总能收到事件，而不管用户当前触摸区域是空白的，还是有其他组件已经处理了触摸事件。

5.2.5 View 组件的其他属性

`removeClippedSubviews` 是布尔类型的属性。它是一个特殊的与性能优化相关的属性，通常在 `ScrollView` 组件或者 `ListView` 组件中使用。当组件有很多子组件不在屏幕显示范围内时，将这个属性设为 `true` 允许释放这些不在显示范围内的子组件以优化性能。

要让此属性生效，组件与子组件的样式键 `overflow` 都应当设置为 `hidden`（默认值）。

`renderToHardwareTextureAndroid`、`needsOffscreenAlphaCompositing` 是 Android 平台独有的两个布尔类型的属性，它们的设置与动画效果有关。本章暂不讨论。

`shouldRasterizeIOS` 是 iOS 平台独有的布尔类型的属性，其设置与动画效果有关。本章暂不讨论。

React Native 有一套属性是专门为了让失能人士（指因为伤病而导致视力或者触摸能力受损的人）更方便地使用手机而准备的，其中大部分属性都是在 `View` 组件中设置的。

这些属性对应的实现目前还处于开发中。这里只是简单地列出它们的名称，不做详细讨论，有需要的读者请参阅最新的 React Native 官方文档。

iOS 和 Android 两个平台都可以使用的属性有：`accessible`、`accessibilityLabel`。

iOS 平台独有的属性有：`accessibilityTraits`、`onAccessibilityTap`、`onMagicTap`。

Android 平台独有的属性有：`accessibilityComponentType`、`accessibilityLiveRegion`、`importantForAccessibility`。

5.2.6 设备放置状态、根 View 与 onLayout 回调函数

移动应用的一大特点就是设备（手机或者 PAD）可以感应到它是被横置的还是竖置的，然后自动调整屏幕的显示方式来适应当前的放置状态。React Native 初始化的项目，默认在两个平台上都能自动调整应用的显示方式来适应设备当前的放置状态（第 18 章介绍了开发者如何将应用设置为始终竖向显示或者始终横向显示）。开发者为了开发这种应用，需要：

- （1）在应用启动时能检测到设备是横置的还是竖置的；
- （2）当设备从横置变为竖置，或者竖置变为横置时，应用要能监测到这个事件。

检测设备当前是竖置的还是横置的一个方法是取当前设备屏幕的宽与高，正常的设备在竖置时，宽小于高；而横置时，宽大于高。

在第 2 章中，我们就已经学习了如何使用 `Dimensions` API 来得到手机屏幕的宽和高。`Dimensions` API 的缺陷（直到 0.20.0 版本都还存在）是它可以取到设备屏幕的宽和高，但始终是应用启动时的宽和高。比如在设备被横置时启动了应用，那么通过 `Dimensions` API 取到的宽大于高；将设备竖置时，这时通过 `Dimensions` API 取到的设备屏幕的宽和高与横置时取到的宽、高值一样。

通常，React Native 开发的应用有一个或者多个根 View。根 View 的特点是它没有父组件。在 React Native 开发中，通过指定这个根 View 组件的 onLayout 回调函数可以很方便地得到初始设备的放置状态，监测设备放置状态的改变并得到改变后新的屏幕高度与宽度。为了实现这些功能，开发者不能指定根 View 的宽和高，并需要设定根 View 组件的样式键 flex 值为 1。代码 5-7 示范了 onLayout 回调函数的使用。

代码 5-7，index.ios.js 或者 index.android.js:

```
'use strict';
var React = require('react-native');
var {
  AppRegistry, StyleSheet, Text, View,
} = React;
var Project19 = React.createClass({
  _onLayout:function(event) {
    {
      //这里使用大括号是为了将 let 解构赋值得到的变量的作用域限制
      //在大括号内，因为接下来还要解构赋值一次
      //使用解构赋值取得设备放置方式被改变后的宽、高与左上角坐标
      let {x,y,width,height} = event.nativeEvent.layout;
      console.log('width from View onLayout:' + width);      //打印宽
      console.log('height from View onLayout:' + height);    //打印高
      console.log('x from View onLayout:' + x);              //打印组件左上顶点的横坐标
      console.log('y from View onLayout:' + y);              //打印组件左上顶点的纵坐标
    }
    //使用解构赋值取得设备屏幕的宽和高，与 onLayout 函数上报的宽、高做比较
    let Dimensions = require('Dimensions');
    let {width,height} = Dimensions.get('window');
    console.log('width from Dimensions:' + width);
    console.log('height from Dimensions:' + height);
    console.log('\r\n');
  },
  _onLayoutText:function(event) {      //Text 组件的 onLayout 回调函数
    let {x,y,width,height} = event.nativeEvent.layout;
    console.log('width from Text onLayout:' + width);
    console.log('height from Text onLayout:' + height);
    console.log('x from Text onLayout:' + x);
    console.log('y from Text onLayout:' + y);
    console.log('\r\n');
  },
  render: function() {
    return (
      <View style={styles.container}
        onLayout={this._onLayout}>      //为根 View 加上 onLayout 回调函数
        <Text style={styles.welcome}
          onLayout={this._onLayoutText}>  //为 Text 组件加上 onLayout 回调函数
          Welcome to React Native!
        </Text>
      </View>
    );
  }
});
var styles = StyleSheet.create({
  container: {
```

```
flex: 1, //将根 View 的 flex 样式键设为 1, 并且不指定根 View 的宽和高
justifyContent: 'center',
alignItems: 'center',
backgroundColor: '#F5FCFF',
},
welcome: {
  fontSize: 20,
  textAlign: 'center',
  margin: 10,
}
});
AppRegistry.registerComponent('Project19', () => Project19);
```

代码 5-7 用于测试设备能改变放置状态。使用真实设备时, 需要注意 Android 和 iOS 平台设备都有禁止屏幕旋转选项, 开发者在测试前需要先关闭禁止屏幕旋转选项。

使用 iOS 模拟器可以在选中模拟器(点击模拟器窗口的上窗框)后, 在屏幕最上方的 Simulator 菜单中选择 Hardware 菜单项, 再选择 Rotate Left、Rotate Right 菜单项改变模拟显示方式, 选择 Shake Gesture 菜单项模拟手机被摇晃事件。操作如图 5-14 所示。

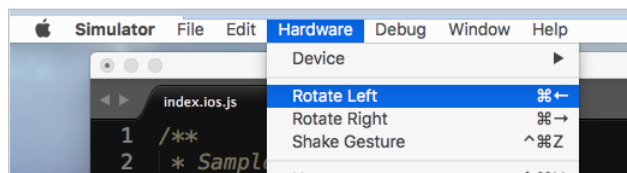


图 5-14 设置 iOS 模拟器模拟真机物理事件

模拟器显示方式被改变时不会触发 Layout 事件, 因此代码 5-7 必须在真实设备上运行进行测试。

代码 5-7 运行在各平台时, 在根 View 的 `_onLayout` 函数中取到的 `height` 是屏幕总高度, 是没有去除状态栏高度的值。

将手机竖置(下次再横置)启动应用, 运行代码 5-7, 可以在日志中看到打印出来的 4 个值。然后改变设备的放置状态, 再查看日志中打印出来的 4 个值。

也许有读者会想: 将应用的显示方式设置为跟随设备放置状态的改变而改变, 却又在根 View 组件中设定了固定的宽、高值, 怎么处理? **答案是: 不要犯这个错误。**

绝大部分 React Native 组件都继承了 View 组件的属性, 这意味着绝大部分组件的 `onLayout` 回调函数都可以向开发者提供该组件的宽、高与组件左上角的坐标值。在代码 5-7 中, 我们给 Text 组件也赋予了 `onLayout` 回调函数, 可以在日志输出中观察到当设备放置状态改变时, Text 组件的位置改变。

`onLayout` 回调函数在组件被加载或者布局被改变时会被调用。我们在后面会看到, 不仅根 View 组件的布局可能会因为手机放置状态的改变而被改变, 其他组件也会因为一些因素而导致改变。最常见的就是 Text 组件, 当用户输入的字符串显示长度超过 Text 组件的长度时, Text 组件有可能会增加一行, 这时它的高度就会发生变化, 从而触发 `onLayout` 事件。

需要注意的是 onLayout 事件（组件被加载或者组件布局被改变的事件）会在布局被 React Native 框架计算好后马上发出，此时新的布局可能还没有被渲染到屏幕上（特别是布局改变带有动画效果时）。

5.2.7 pointerEvents 属性

在 React Native UI 开发中，很多组件被布局在手机屏幕上，其中有一些组件使用绝对定位布局，在代码运行时的某个时刻有可能会遮盖住它的位置下方的某个组件的部分或者全部。

提示：绝对定位只是说这个组件的位置由它距父组件的边框距离决定。这个组件距父组件边框的距离、宽、高值在应用运行期间还是可以改变的。

在 React Native 框架中，触摸事件总是被传送给最上层的组件。在 render 函数的 JSX 代码中，子组件可以被认为放置在父组件的上方，绝对定位的组件可以被认为会覆盖在它前面布局（JSX 代码中的顺序）的组件的上方。对于某些应用逻辑，被遮盖住的组件需要处理触摸事件。比如我们有一个地图组件上覆盖了一个图像组件用来显示信息，但又不想让这个图像组件影响用户手指拖动地图的操作，这时就需要使用图像组件从 View 组件继承得到的 pointerEvents 属性来解决这个问题。

pointerEvents 是字符串类型的属性，它可以取值为 none、box-none、box-only、auto。当取值为 none 时，发生在本组件与本组件的子组件上的触摸事件都会交给本组件的父组件处理。当取值为 box-none 时，发生在本组件显示范围内（但非本组件的子组件显示范围内）的事件将交由本组件的父组件处理，发生在本组件的子组件显示范围内的触摸事件由子组件处理。当取值为 box-only 时，发生在本组件显示范围内的触摸事件将全部由本组件处理（即使触摸事件发生在本组件的子组件显示范围内）。而 auto 的行为视组件的不同而不同，并不是所有的子组件都支持 box-none 和 box-only 两个值，开发者在开发时需要自行测试。

开发者可以在代码中按业务逻辑修改一个组件的 pointerEvent 属性值为上述 4 个值，还可以在需要恢复默认触摸事件处理逻辑时将组件的 pointerEvents 属性值修改为 null。

从上述讨论中可以看到，这个属性虽然是在 View 组件中定义的，但目的是给从 View 组件继承的其他组件使用。代码 5-8 作为 pointerEvents 的一个简单示例，演示了如何让一个按钮能触摸或者不能触摸（本例中使用 Text 组件作为最简单的按钮，但对所有其他可触摸组件道理都是相通的）。

代码 5-8，index.ios.js 或者 index.android.js：

```
'use strict';
var React = require('react-native');
var {
  AppRegistry, StyleSheet, Text, View, StatusBarIOS
} = React;
var Project19 = React.createClass({
  getInitialState:function() {
    return {
      bigButtonPointerEvents:null //状态机变量控制大按钮是否工作
```

```

    };
  },
  onBigButtonPressed:function() {
    console.log('Big button pressed');
  },
  onSmallButtonPressed:function() {
    if ( this.state.bigButtonPointerEvents === null ) {
      console.log('big button will not responde. ');
      this.setState({bigButtonPointerEvents:'none'});    //改变状态机变量值
      return;
    }
    console.log('big button willresponde. ');
    this.setState({bigButtonPointerEvents:null});          //改变状态机变量值
  },
  render: function() {
    return (
      <View style={styles.container}>
        <Text style={styles.sButtonStyle}
          onPress={this.onSmallButtonPressed}>
          Small button
        </Text>
        <Text style={styles.bButtonStyle}
          onPress={this.onBigButtonPressed}
          pointerEvents={this.state.bigButtonPointerEvents}>    //设置属性
          Big button
        </Text>
      </View>
    );
  }
});
var styles = StyleSheet.create({
  container: {
    flex: 1
  },
  sButtonStyle: {
    fontSize:20, left:130, top:50, width:150, height:35, backgroundColor:'grey'
  },
  bButtonStyle: {
    fontSize:20, left:130, top:130, width:150, height:70, backgroundColor:'grey'
  }
});
AppRegistry.registerComponent('Project19', () => Project19);

```

运行代码 5-8 后，手机屏幕上会有两个按钮。最开始时，正文的 Big button 可以正常工作，点击后会在日志中打印“Big button pressed”；上方的小按钮，按一下会让大按钮不处理触摸事件，再按一下会让大按钮正常工作。代码 5-8 运行日志输出参见图 5-15。

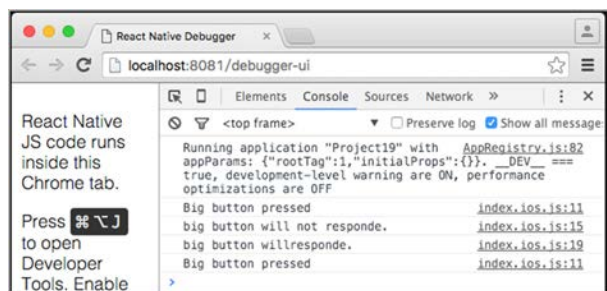


图 5-15 代码 5-8 运行日志输出

5.3 Image 组件

React Native 可以从给定的网址、给定的本地文件或者项目的资源文件中加载图片。

当我们决定在 UI 上的某个位置显示一张图片时,首先需要确定这张图片是 JPG 格式还是 PNG 格式的。PNG 格式的优点是有透明度设置,可以将 PNG 图片下的图像显示出来。因此,当需要这个功能时,就要使用 PNG 图片,否则使用 JPG 图片。显示 JPG 图片的速度要快于 PNG 图片。这也是很好理解的,因为不需要处理透明度。JPG 图片的这种优势,只有在比较差的手机上,并且 UI 上有比较多的图片时才会被用户观察到,因此开发者也无须太关心这一点。

在图像处理中,使用 SVG 图片是比较先进的技术,它可以实现清晰的图片缩放,但目前 React Native 还不能支持 SVG 图片。一个变通的办法是在 React Native 的 WebView 组件中载入 SVG 图片(将在后续章节中讨论)。

5.3.1 加载网络图片

在 React Native 开发中,加载网络图片的方法非常简单,示例如代码 5-9 所示。

代码 5-9:

```
.....
var imageAddress = 'http://h.hiphotos.baidu.com/image/xxxx';
.....
return (
  <View style={styles.container}>
    <Image
      style={styles.imageStyle}
      source={{uri: imageAddress}} />
    </View>
  .....
)
```

imageAddress 中的 xxxx 需要替换为任意一张真实图片的地址。也可以不使用字符串变量,而使用 source={{uri: 'xxxx'}} , 其中的 xxxx 替换为任意一张真实图片的地址。

在 iOS 平台上,开发者可以在没显示图片之前,先使用 Image 组件的静态函数 getSize 取得指定 URI 地址图片的宽和高。取宽、高的代码参见代码 5-10。

代码 5-10:

```
.....
componentDidMount: function() {
  Image.getSize( aURI ).then( (width, height)=> {
    ..... //正确取到宽和高,进行相应处理
  }).catch( (error)=> {
    console.log( error ); //取宽、高出错
  });
},
.....
```

代码 5-10 使用了 JavaScript 的 Promise 机制。不熟悉的读者可以参阅附录 A.5。

在 React Native 0.20.0 版本中,在 iOS 平台上代码调用 getSize 函数取图片的宽、高,React Native

框架事实上会下载这张图片，并且将该图片保存在缓存中。所以 `getSize` 函数可以作为预加载图片资源的一个方法。但 `Image` 组件仍然在调研演进之中，图片预加载有可能会演变为一个跨平台的静态函数。请大家关注 `Image` 官方文档，获取相关的最新信息。

5.3.2 加载静态图片资源

在 React Native 开发中，需要预先加载静态的图片资源，如代码 5-11 所示。

代码 5-11:

```
<View style={styles.container}>
  <Image style={styles.icon}
    source={require('./image/redicon.png')} />
</View>
```

在上面的语句中，`require` 等同于使用 `var` 声明了一个变量，因为 `var` 的变量提升效应，等同于在代码顶部预先加载了图片资源。

代码 5-12 在与上面示例同样的运行环境中运行时将会出错。

代码 5-12:

```
.....
var imageName = './image/redicon.png';
.....
<View style={styles.container}>
  <Image style={styles.icon}
    source={require({imageName})} />
</View>
.....
```

在 React Native 开发中，不允许使用字符串变量来指定需要预先加载的图片名称。因为 React Native 在编译代码时处理所有的 `require` 声明，还不是在运行时动态的处理，所以不能动态加载图片资源。

5.3.3 加载资源文件中的图片

React Native 可以加载 Android 项目或者 iOS 项目中的图片资源文件。但本书不打算讨论如何加载，原因有两个。

(1) React Native 代码所使用的静态的图片资源文件即使与项目的资源文件有重复，也建议单独存放，不要共用。这样代码结构更清晰。

(2) React Native 在加载资源文件中的图片时，使用的是不检查机制。也就是说，在编译代码时不会去检查资源图片是否真的存在，有可能发生在代码运行到需要取资源文件中的图片时，才发现该图片不存在的情况，代码容易出错。

5.3.4 动态加载手机中的图片资源

在 React Native 0.19.0 版本中，`CameraRoll` API 终于实现了对 Android 平台的支持。通过

CameraRoll 组件，React Native 应用可以读取手机中存储的图片资源。CameraRoll API 比较复杂，它的相关知识将在第 14 章中讨论。

React Native 还支持加载以 Base64 编码格式保存的图片。相关内容也在第 14 章中讨论。

5.3.5 Image 组件的样式

Image 组件必须在样式中声明图片的宽和高。如果没有声明，则图片将不会被呈现在界面上。

图片样式名称可以使用 5.1 节中讨论的所有 flexbox 样式名称。除此之外的样式名称将在下面讨论。

如果我们知道需要显示的图片分辨率，那么就可以在手机上做到点对点的显示图片。为了精确地显示图片，假设图片的实际分辨率为 `actualWidth × actualHeight`，定义图片显示样式的代码参见代码 5-13。

代码 5-13:

```
.....
let PixelRatio = require('PixelRatio');
let pixelRatio = PixelRatio.get();
.....
perciseImageStyle: {
  width: actualWidth/pixelRatio,
  height: actualHeight/pixelRatio,
},
.....
```

在下面的讨论中，我们将 Image 定义的宽、高乘以当前运行环境的像素密度称为 Image 的实际宽、高。当 Image 的实际宽、高与图片的实际宽、高不符时，视图片样式定义中 `resizeMode` 的取值不同而分为三种情况。`resizeMode` 的三个取值分别是：`contain`、`cover` 和 `stretch`。如果没有在样式中定义 `resizeMode`，那么其默认取值为 `cover`。但不论 `resizeMode` 取何值，图片都会在 Image 组件的显示区域居中显示。也就是说，显示出来的图片的中点与显示区域的中点是一个点。

5.3.5.1 cover 模式

`cover` 模式图片处理的思路是要求填充整个 Image 定义的显示区域，可以对图片进行放大或者缩小，可以丢弃放大或者缩小后的图片中的部分区域，只求在显示比例不失真的情况下填充整个显示区域。

如果图片的宽、高有一个值小于 Image 的实际宽、高，React Native 会对图片进行放大，直到图片的宽与高均不小于 Image 的实际宽、高，然后将放大的图片居中显示，超出显示区域的部分被直接丢弃。

如果图片的宽、高均大于 Image 的实际宽、高，React Native 会对图片进行等比缩小，直到缩小后图片的宽、高有一个值等于 Image 的实际宽、高，然后将缩小后的图片居中显示，超出显示区域的部分被直接丢弃。

5.3.5.2 contain 模式

contain 模式图片处理的思路是要求显示整张图片，可以对它进行等比缩小，但不能丢弃缩小后图片的某部分。它的详细处理如下：

如果图片的实际宽、高都小于 Image 的实际宽、高，React Native 不会对图片进行任何缩放，只是把它居中呈现在父 View 中。在这种情况下，图片得到无失真的呈现，但图片无法填充 Image 的所有区域，会在图片四周留下 Image 组件的底色。

如果图片的实际宽、高有一个值或者都大于 Image 的实际宽、高，React Native 会对图片进行等比缩小，直到缩小后图片的宽、高都不大于 Image 的实际宽、高，然后在 Image 中展现图片。如果 Image 的宽、高比与图片的宽、高比不相等，则会在左、右侧或者上、下侧留下两条空白，由 Image 组件的底色填充。

5.3.5.3 stretch 模式

stretch 模式图片处理的思路是要求填充整个 Image 定义的显示区域，为此按照需要对图片进行任意的缩放，不考虑保持图片原来的宽、高比。这种模式显示出来的图片有可能会明显的失真。

5.3.5.4 其他样式键

Image 组件还支持 backgroundColor、borderColor、borderWidth、overflow、opacity 样式键。这些键都已经在 View 组件中讨论过，这里不再赘述。

类似于前面在讨论 View 组件时提到的，borderRadius 可以实现图片的圆角效果，甚至将一张正方形的图片显示为一张圆形的图片（丢弃部分图片）。

tintColor 是 iOS 平台的专有属性，颜色类型，可以让图片中的非透明像素部分有一种被染色的效果。

overlayColor 是 Android 平台的专有属性，颜色类型。在 Android 平台上，在某些情况下无法通过 borderRadius 实现圆角效果，这时需要使用 overlayColor 属性，强行将需要圆角的部分使用指定的颜色填充，从而实现圆角效果。

5.3.6 Image 组件显示特性

图 5-16 是一张分辨率为 600 × 360 的图片。在笔者测试用的 Android 手机与 iOS 6 Plus 模拟器中，像素比例都为 3，完全点对点显示这张图片需要一个宽、高为 200 × 120 的 Image 组件。接下来，我们分情况使用三种 resizeMode 样式在手机上显示这张图片。



图 5-16 图片原图

测试 Image 组件显示特性的代码见代码 5-14。

代码 5-14, index.ios.js 或者 index.android.js:

```
'use strict';
var React = require('react-native');
var {
  AppRegistry, StyleSheet, View, Image
} = React;
var Project19 = React.createClass({
  imageA:null,
  componentWillMount:function() {
    this.image1 = require('./girl.jpg');
  },
  render: function() {
    return (
      <View style={styles.container}> //接下来以三种不同的模式分别显示图片
        <Image style={[styles.imageStyle,{resizeMode:'cover'}]}
          source={this.image1}/>
        <Image style={[styles.imageStyle,{resizeMode:'contain'}]}
          source={this.image1}/>
        <Image style={[styles.imageStyle,{resizeMode:'stretch'}]}
          source={this.image1}/>
      </View>
    );
  }
});
var styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    borderWidth:1,
    backgroundColor: 'grey',
  },
  imageStyle: {
    margin:2, backgroundColor: 'white',
    height:100, width:100 }
});
AppRegistry.registerComponent('Project19', () => Project19);
```

5.3.6.1 Image 组件宽、高都小于所需宽、高

代码 5-14 运行效果参见图 5-17。从上到下，三张图片分别是 cover、contain 和 stretch 模式。当 Image 组件的宽、高都小于所需宽、高时，在两个平台上显示没有区别，因此只给出一个平台效果图。

5.3.6.2 Image 组件宽、高都大于所需宽、高

将代码 5-14 倒数第三行的宽、高值均修改为 220，再次运行后效果参见图 5-18。在 Android 平台上，上、下两张图片都已经有一小部分超出屏幕范围（有兴趣的读者可以给根 View 与 Image 都加个 onLayout 回调函数，看看这时的各个值）。cover 模式与 stretch 模式在两个平台上的显示特性没有区别，但 contain 模式有区别。

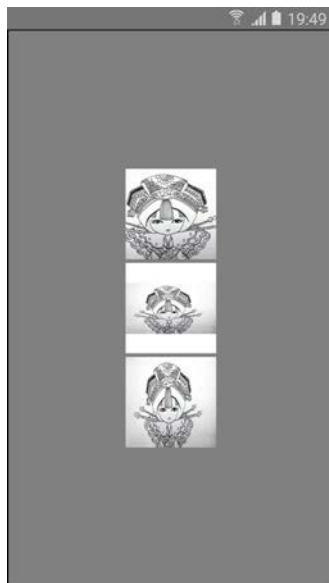


图 5-17 代码 5-14 运行效果，宽、高均为 100

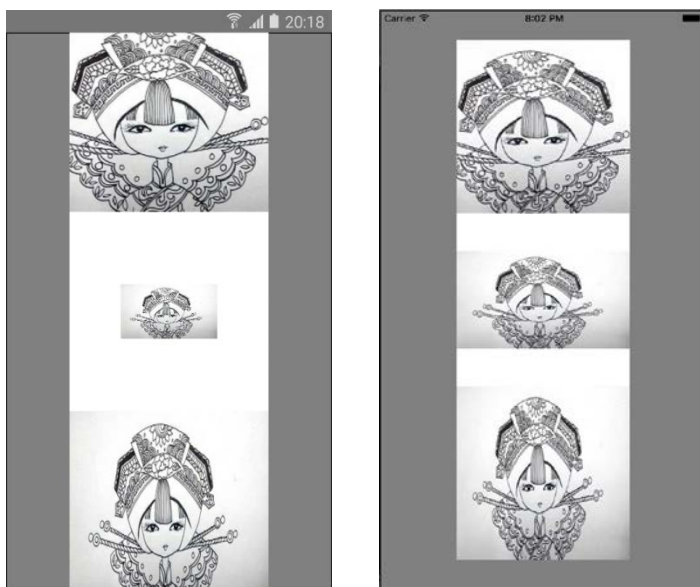


图 5-18 代码 5-14 运行效果，宽、高均为 220

5.3.6.3 Image 组件宽小于所需宽、高大于所需高

将代码 5-14 倒数第三行的宽、高值均修改为 150，再次运行后效果参见图 5-19。cover 模式与 stretch 模式在两个平台上的显示特性没有区别，但 contain 模式有区别。

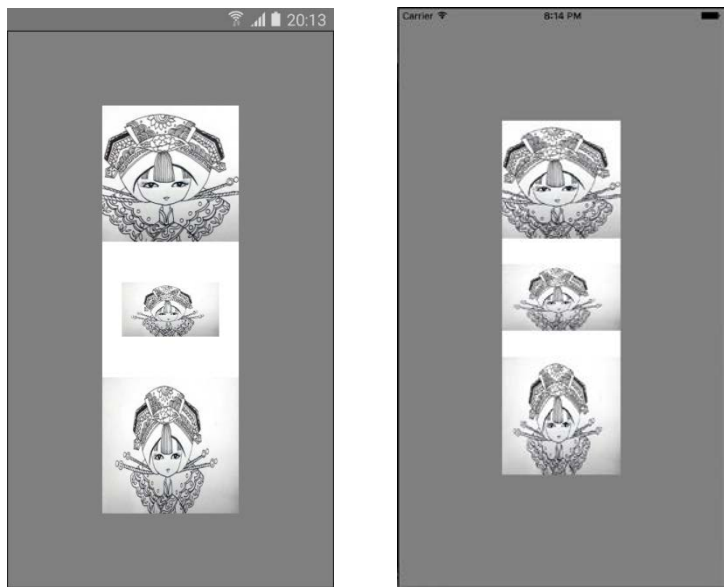


图 5-19 代码 5-14 运行效果，宽、高均为 150

5.3.7 Image 组件的其他属性

在通过 Image 组件读取并显示网络上图片资源时,我们可以通过设置 `onLoadStart`、`onLoadEnd`、`onLoad` 三个属性来指定在开始读取与读取结束时所需要进行的函数处理。`onLoad` 属性指定的回调函数只有在正确读取到图片资源时才会被执行,而 `onLoadEnd` 函数不论读取是否成功都会被执行。

这三个回调函数都会带有一个 `event` 参数,但 `event` 参数中只有一个 `event.timeStamp` 对开发者有用,它记录了事件发生的时间。

虽然 Image 不是从 View 组件继承而来的,但它还是支持 `onLayout` 回调函数,使用方法与 View 组件的 `onLayout` 函数类似。

5.4 可触摸组件

移动应用程序存在 Web App 与 Native App 之分。Web App 是指用 HTML 语言写出来的网页被封装在手机的网页浏览器中而得到的应用;Native App 是指用原生开发语言开发的应用。Web App 给用户的体验没有 Native App 好,主要的表现就是在对用户触屏事件的反应上。Native 在用户触摸屏幕时,被触摸的 UI 组件会发生视觉上的变化(比如某项变暗或者高亮,长按会出现弹出菜单等);而 Web App 很难做到这一点。React Native 为了实现这种体验而提供了可触摸组件。

可触摸组件并不是一个单独的组件,它需要与其他组件协同工作。通过将其他组件定义为可触摸组件的子组件,我们将该组件变成了一个可触摸组件。当用户触摸它时,会发生相应的视觉变化,同时开发者可以处理触摸事件。

为了使代码简洁,本书中需要按钮的地方都用一个很丑陋的 Text 组件代替了。本书中,介绍

可触摸组件的篇幅也不多（因为它是一个比较简单的组件）。但在真正商用开发时，代码中会大量用到可触摸组件，使用它配合美工给出的图片，能做出精美的按钮等触摸反馈视觉效果。

5.4.1 可触摸组件类型

可触摸组件有 `TouchableNativeFeedback`、`TouchableWithoutFeedback`、`TouchableHighlight` 和 `TouchableOpacity` 四种。

`TouchableNativeFeedback` 是 Android 操作系统专用组件。它使用原生视图的相应状态来展示触摸事件的视觉效果。因为这是一个 Android 独有的组件，并且没有提供什么特殊的 UI 效果，使用面非常窄。本书中不讨论这个组件。

`TouchableWithoutFeedback` 组件在用户触摸时，没有反馈任何视觉效果，但可以让 React Native 处理触摸事件。除非开发者特意需要这种效果，否则不要使用这个组件。

`TouchableOpacity` 与 `TouchableHighlight` 两个组件在用户触摸时都反馈了视觉效果（将在后面详细讨论）。在这里读者可以记住一点，当没有特殊要求时，优先考虑使用 `TouchableOpacity` 组件。

可触摸组件只能有一个子组件。如果需要将多个组件设置为可触摸效果，则需要将它们放在一个 View 中，然后让这个 View 成为可触摸组件的子组件。

5.4.2 TouchableOpacity 组件

当一个组件成为 `TouchableOpacity` 组件的子组件后，这个组件被触摸时会变成半透明的组件。也就是说，这个组件遮盖的背景颜色图案将会透过它被显示出来。`View` 组件、`Image` 组件、`Text` 组件都可以成为它的子组件。通常它的声明如下：

```
<TouchableOpacity onPress={this._onPressButton}
  activeOpacity={ XXX } >
```

其中，`activeOpacity` 定义了透明度的值，取值为从 0 到 1。0 表示完全透明，1 表示一点也不透明。它的默认值是 0.2。普通开发者不需要改动 `activeOpacity` 的默认值。

5.4.3 TouchableHighlight 组件

当一个组件成为 `TouchableHighlight` 组件的子组件后，这个组件被触摸时会产生一种变暗的效果。

`TouchableHighlight` 在某些情况下会发生显示异常。代码 5-15 示范了这一点。在运行这段代码前，请在当前目录下随便放置一张图片，JPG 或者 PNG 格式都可以，保持文件名称与代码中文件名称相同。

代码 5-15：

```
'use strict';
var React = require('react-native');
```



```

var {
  AppRegistry, StyleSheet, Text, View,TouchableHighlight
} = React;
var Project19 = React.createClass({
  render: function() {
    return (
      <View style={{flex: 1, backgroundColor: 'white'}}>
        <TouchableHighlight>
          <View style={{width:120,height:70,backgroundColor:'grey'}}/>
        </TouchableHighlight>
      </View>
    );
  }
});
AppRegistry.registerComponent('Project19', () => Project19);

```

代码 5-15 运行效果如图 5-20 所示。未按下按钮时的应用显示如左图所示，而按下按钮时的应用显示如右图所示，可以看到右图中按钮右侧不应当变暗的地方受到了 TouchableHighlight 组件的影响。

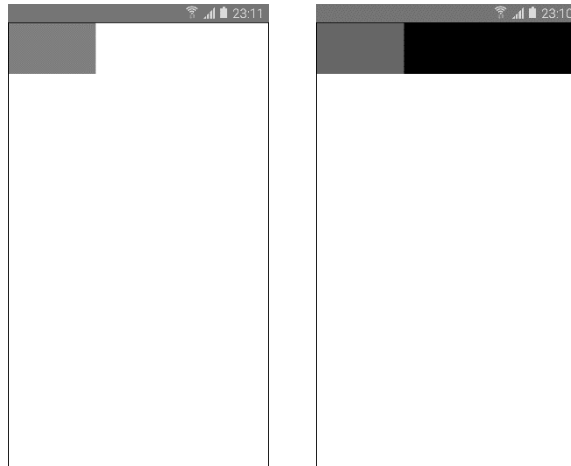


图 5-20 代码 5-15 运行效果

这个问题的解决办法也很简单，可以在这张图片的右侧再增加一些其他需要显示的组件，或者为整个背景加一张图片。

TouchableHighlight 使子组件变暗的思路是让被子组件遮盖住的下一层颜色向上透出来，这样来使子组件变暗或者染色。实际的实现是在需要时，在当前的 View 结构上再加一层 View，不需要时去掉这层 View。这无疑是比较复杂的操作。这也是为什么一开始说优先考虑使用 TouchableOpacity 的原因。

TouchableHighlight 组件也有 activeOpacity 属性，默认值是 0.8。

TouchableHighlight 有三个独有的属性：

- onShowUnderlay, 回调函数，当下层开始被展现时（也就是被触摸时）被调用。
- onHideUnderlay, 回调函数，当下层不再被展现时被调用。

- `underlayColor`，字符串，指定下层颜色。

5.4.4 其他属性

`TouchableHighlight` 组件和 `TouchableOpacity` 组件都可以接受 `TouchableWithoutFeedback` 组件的属性，因此在此一起讨论。

`onPress` 属性是一个回调函数。`onPress` 事件会在手指离开组件后马上被激活。需要注意的是，在手指停留在组件上期间，组件的触摸响应者身份被取消，`onPress` 事件将不会被激活。

`onPressIn` 属性是一个回调函数。`onPressIn` 事件会在手指接触组件 `delayPressIn` 毫秒后马上被激活。

`onPressOut` 属性是一个回调函数。`onPressOut` 事件会在手指接触组件 `delayPressOut` 毫秒后马上被激活。

`delayLongPress` 用来设置按了多少毫秒后，`onLongPress` 事件会被激活。如果没有设置，默认是 500ms。

`delayPressIn` 用来设置手指接触屏幕多少毫秒后，`onPressIn` 事件会被激活。默认值是 0。

`delayPressOut` 用来设置手指离开屏幕多少毫秒后，`onPressOut` 事件会被激活。默认值是 0。

`pressRetentionOffset` 是对象类型的属性。它的格式是：

```
{
  top: number,
  left: number,
  bottom: number,
  right: number
}
```

只能在当前可触摸组件的父组件不可滚动的情况下设置这个属性。

这个属性决定了当手指移开距组件多远距离之后，可触摸组件会变成不被触摸的状态。如果手指再次进入这个范围内，可触摸组件会再次变成触摸状态。为这个参数传入一个常量对象可以减少内存消耗。

在当前视图不能滚动的前提下指定这个属性，可以决定当手指移开多远距离之后，会不再激活按钮。如果手指再次移回这个范围内，按钮会被再次激活。只要视图不能滚动，就可以来回进行多次这样的操作。确保传入一个常量来减少内存消耗。

`onLayout` 回调函数这里不再讨论。

5.5 加深理解三大组件

`View`、`Image` 和 `Touchable` 组件在 React Native 开发中占据非常重要的地位，本节再讨论几个例程，以加深对这三大组件的理解。

5.5.1 使用导航栏的导航框架

现在我们讨论到的 React Native 组件已经足够搭建一个使用导航栏的导航框架了。

导航栏通常位于手机屏幕的顶部或者底部，它通常有 3 个或者 4 个栏目，对应着 3 张或者 4 张图片（用来介绍每个栏目）。我们将要实现的例程（见代码 5-16）运行效果如图 5-21 所示。

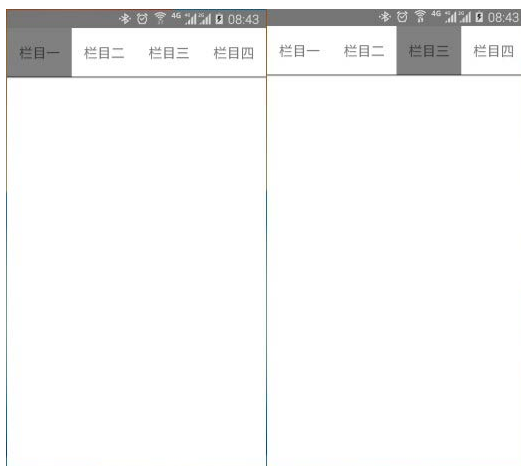


图 5-21 代码 5-16 运行效果

左、右两图分别显示了栏目一和栏目三被选择时的情况。虽然在导航栏下方都有一个空白的 View，但它们是两个不同的 View。

也许有读者会问，第 5 章都要结束了，为什么实现的界面还是这么朴素？让界面美观需要优秀的美工，只要有合适的图片与配色，上图所示的界面就可以从朴素变为美观。因为本书是黑白印刷，因此在屏幕截图颜色选择上只能是白色、黑色、灰色三种，UI 界面想花哨都花不起来啊。

5.5.1.1 导航栏自定义组件的实现

我们首先需要知道应用有多少个栏目，以及每张导航图片的宽、高比。在这里，假设应用有 4 个栏目，宽、高比为 4:3。

自定义组件 NavBar 在 NavBar.js 文件中实现，参见代码 5-16。

代码 5-16，NavBar.js:

```
'use strict';
var React = require('react-native');
var Dimensions = require('Dimensions');
var totalWidth = Dimensions.get('window').width;           //在这里计算导航栏按钮的宽与高
let naviButtonWidth = totalWidth / 4;
let naviButtonHeight = naviButtonWidth * 0.75;
var {
  AppRegistry, StyleSheet, Text, View, TouchableHighlight,
} = React;

var NavBar = React.createClass({                          //声明两个必须要有的属性
```

```

propTypes: {
  naviBarStatus: React.PropTypes.arrayOf(React.PropTypes.number).isRequired,
  onNaviBarPress: React.PropTypes.func.isRequired,
},
//四个按钮被按下时的处理函数，在处理函数中调用回调函数
_naviTab0Pressed: function() {
  this.props.onNaviBarPress(0);
},
_naviTab1Pressed: function() {
  this.props.onNaviBarPress(1);
},
_naviTab2Pressed: function() {
  this.props.onNaviBarPress(2);
},
_naviTab3Pressed: function() {
  this.props.onNaviBarPress(3);
},
render: function() {
  //通过属性得知哪个导航按钮是当前的导航页，这个导航按钮将用灰色背景
  //利用 JavaScript 数组的 map 函数，从一个数组对应生成另一个数组 buttonColors
  //这一步不是必需的，可以用其他方法来实现，但这样写使代码结构清晰，便于理解
  //对 JavaScript 数组的 map 函数用法不熟的读者请参见附录 A.6
  var buttonColors = this.props.naviBarStatus.map(function(aNumber) {
    if (aNumber == 0) return 'white';
    return 'gray';
  });
  return (
    <View style={styles.naviRow}>
      <TouchableHighlight onPress={this._naviTab0Pressed}>
        <View style={[styles.button, {backgroundColor:buttonColors[0]}]}>
          <Text style={styles.textStyle1}>
            栏目一
          </Text>
        </View>
      </TouchableHighlight>
      <TouchableHighlight onPress={this._naviTab1Pressed}>
        <View style={[styles.button, {backgroundColor:buttonColors[1]}]}>
          <Text style={styles.textStyle1}>
            栏目二
          </Text>
        </View>
      </TouchableHighlight>
      <TouchableHighlight onPress={this._naviTab2Pressed}>
        <View style={[styles.button, {backgroundColor:buttonColors[2]}]}>
          <Text style={styles.textStyle1}>
            栏目三
          </Text>
        </View>
      </TouchableHighlight>
      <TouchableHighlight onPress={this._naviTab3Pressed}>
        <View style={[styles.button, {backgroundColor:buttonColors[3]}]}>
          <Text style={styles.textStyle1}>
            栏目四
          </Text>
        </View>
      </TouchableHighlight>
    </View>
  );
}

```

```

        </TouchableHighlight>
      </View>
    );
  }
});
var styles = StyleSheet.create({
  naviRow: {
    flexDirection: 'row',
  },
  button: {
    width: naviButtonWidth,
    height: naviButtonHeight,
    justifyContent: 'center',
  },
  textStyle1: {
    fontSize: 20,
    textAlign: 'center',
  },
});
module.exports = NavBar;

```

在这里，请读者注意 `NaviBar` 自定义组件使用到的样式。只定义了按钮的宽、高，而它的父 `View` 没有定义宽、高。在这种情况下，父 `View` 的高由其子组件来决定。

在这个实现中，我们是简单地在每个长方形的 `View` 中放入不同的文字做成一个导航按钮，然后通过定义 `View` 的不同背景色来区分导航按钮的状态。通过本章介绍的知识，读者应当有足够的知识将 4 个 `View` 换成 4 个 `Image`，然后通过对 `Image` 定义透明度或者利用被染色之类的机制来区分它们的状态。用 `Image` 组件做导航栏会非常美观，但前提是请美工给你提供 4 张图片。

5.5.1.2 调用自定义组件

`index.android.js` (或者 `index.ios.js`) 文件内容如下(事实上这部分代码比第 3 章的对应代码还简单，不需要处理 `Android` 的返回键)。

代码 5-17, `index.android.js` 或者 `index.ios.js`:

```

'use strict';
var React = require('react-native');
var {
  AppRegistry, Navigator,
} = React;
var Page1 = require('./Page1');
var Page2 = require('./Page2');
var Page3 = require('./Page3');
var Page4 = require('./Page4');

var NaviModule = React.createClass({
  configureScene: function(route) {
    return Navigator.SceneConfigs.FadeAndroid;
  },
  renderScene: function(router, navigator) {
    this._navigator = navigator;
    switch (router.name) {

```

```

        case "Page1":
            return <Page1 navigator={navigator}/>;
        case "Page2":
            return <Page2 navigator={navigator} />;
        case "Page3":
            return <Page3 navigator={navigator} />;
        case "Page4":
            return <Page4 navigator={navigator} />;
    }
},
render: function() {
    return (
        <Navigator
            initialRoute={{name: 'Page1'}}
            configureScene={this.configureScene}
            renderScene={this.renderScene} />
    );
}
});
AppRegistry.registerComponent('Project19', () => NaviModule);

```

Page1.js 文件内容参见代码 5-18。

代码 5-18, Page1.js:

```

'use strict';
var React = require('react-native');
var NaviBar = require('./NaviBar');
var {
    StyleSheet, View,
} = React;

var Page1 = React.createClass({
    render: function() {
        //不同的 Page, 需要对应修改下面这个数组
        var naviStatus = [1, 0, 0, 0];
        return (
            <View style={styles.container}>
                <NaviBar naviBarStatus={naviStatus}
                    onNaviBarPress={this.onNaviBarPress}/>
                <View style={styles.whatLeft}/>
            </View>
        );
    },
    //不同的 Page, 需要对应修改下面这个函数中的各返回值
    onNaviBarPress: function(aNumber) {
        switch (aNumber) {
            case 0:
                return;
            case 1:
                this.props.navigator.replace({
                    name: 'Page2',
                });
                return;
            case 2:
                this.props.navigator.replace({

```

```

        name: 'Page3',
      });
      return;
    case 3:
      this.props.navigator.replace({
        name: 'Page4',
      });
      return;
    }
  },
});

var styles = StyleSheet.create({
  container: {
    flex: 1,
  },
  whatLeft: {
    flex: 1,
    borderTopWidth: 1,
    borderColor: 'black',
  },
});
module.exports = Page1;

```

注意到代码中有一个使用 `whatLeft` 定义样式的 `View` 组件，在这个 `View` 中，开发者可以排列各个页面不同的 `React Native` 组件，完成各个页面的功能。为了美观，`whatLeft` 定义样式的 `View` 组件定义了一个上边框。

在这里顺便指出两点：当根 `View` 没有指定背景色时，默认值是一种灰色；而当子 `View` 没有指定背景色时，会继承父 `View` 的背景色。

`Page2.js`、`Page3.js`、`Page4.js` 文件内容不再给出，只是在 `Page1.js` 文件内容中稍做相应的修改，请读者自行完成。

5.5.2 等比放大无丢失显示图片

如果读者足够细心的话，就会发现在 5.3.5 节介绍的三种图片处理方式中，无法实现对图片等比放大后无丢失显示。

或者这么说：有一张 20×10 的图片，要把它放入一个 40×30 的显示区域内。我们可以做到：

- `contain` 模式，图片显示分辨率为 20×10 ，四周都有空白；
- `cover` 模式，图片放大为 60×30 ，然后切成 40×30 ，丢失部分图片内容；
- `stretch` 模式，图片放大为 40×30 ，丢失原始的宽、高比。

但我们无法做到将图片放大为 40×20 ，然后再显示。接下来讨论如何实现这个目标。

... 操作符（也被叫作延展操作符，`spread operator`）已经被 `ES 6` 数组支持。`React` 在它的基础上实现了 `JSX` 语法的展开属性（对此不了解的读者可以参考附录 A.4）。

实现自定义组件 ImageEquallyEnlarge，参见代码 5-19。

代码 5-19, ImageEquallyEnlarge.js:

```
'use strict';
var React = require('react-native');
var {
  StyleSheet, Image,
} = React;
var ImageEquallyEnlarge = React.createClass({
  //ImageEquallyEnlarge 组件的状态机变量是一个 style，它将被用于定义显示图片的样式
  getInitialState: function() {
    return {
      style: {}
    };
  },
  //声明必须有的图片原始宽度与高度
  propTypes: {
    originalWidth: React.PropTypes.number.isRequired,
    originalHeight: React.PropTypes.number.isRequired,
  },
  //此函数被挂接到组件的 onLayout 事件上，当组件被布局时，此函数被调用
  //在此函数中计算新的宽度与高度并将其保存在组件的状态机变量中
  //注意它是如何取得布局的宽、高的
  onImageLayout: function(event) {
    let layout = event.nativeEvent.layout;
    if (layout.width <= this.props.originalWidth) return;
    if (layout.height <= this.props.originalHeight) return;
    let originalAspectRatio = this.props.originalWidth / this.props.originalHeight;
    let currentAspectRatio = layout.width / layout.height;
    if (originalAspectRatio === currentAspectRatio) return;
    if (originalAspectRatio > currentAspectRatio) {
      let newHeight = layout.width / originalAspectRatio;
      this.setState({
        style: {
          height: newHeight,
        }
      });
      return;
    }
    let newWidth = layout.height * originalAspectRatio;
    this.setState({
      style: {
        width: newWidth,
      }
    });
  },
  // {...this.props} 是 JSX 参考的新语法特性之一展开属性 (Spread Attributes)，不熟悉的
  //读者可以参考附录 A.4。使用这个语法将 ImageEquallyEnlarge 组件收到的 props 透
  //传给 Image 组件，然后使用组件的多样式声明来修改原来的样式，实现等比放大
  render: function() {
    return (
      <Image {...this.props}
        style = {[this.props.style, this.state.style]}
        onLayout = {this.onImageLayout}>
    );
  }
});
```



```

    </Image>
  );
}
});
module.exports = ImageEquallyEnlarge;

```

在 ImageEqualEnlarge 组件的 render 函数中，为 onLayout 事件挂接了 onImageLayout 函数。

代码 5-20 是调用自定义组件的示例（读者请自行准备一张 120 × 70 分辨率的图片，或者按照图片实际宽、高修改代码中的 originalWidth 和 originalHeight）。

代码 5-20，index.android.js 或者 index.ios.js：

```

'use strict';
var React = require('react-native');
let ImageEquallyEnlarge = require('./ImageEquallyEnlarge');
var {
  AppRegistry, StyleSheet, View, Image,
} = React;
var Project19 = React.createClass({
  render: function() {
    return (
      <View style={styles.container}>
        < ImageEquallyEnlarge style={styles.image1Style}
          source={require('./tab.jpg')}
          originalWidth={120}
          originalHeight={70}/>
        < ImageEquallyEnlarge style={styles.image2Style}
          source={require('./tab.jpg')}
          originalWidth={120}
          originalHeight={70}/>
      </View>
    );
  }
});
var styles = StyleSheet.create({
  container: {
    backgroundColor: 'blue',
  },
  image1Style: {
    width: 240,
    height: 360,
    backgroundColor: 'red',
  },
  image2Style: {
    width: 300,
    height: 460,
    backgroundColor: 'red',
  },
});
AppRegistry.registerComponent('Project19', () => Project19);

```

5.5.3 宽、高动态变化的组件呈现

在代码 5-20 中，已经展示了如何将宽、高动态变化的组件按照 `flexDirection` 的方向无缝、灵活地呈现在手机屏幕上。第一张图片的高度会被动态地换为 140，而第二张图片的高度会被动态地换为 175，可以看到在两张图片的右侧都会留下比较突兀的背景色。这是例程故意留出来以提醒读者在开发时应当注意的问题。这个问题的解决不在本书讨论范围中，一个好的美工会解决这个问题。

第 6 章

Text、TextInput 等相关知识

Text 组件用来显示字符串，TextInput 组件用来让用户输入字符串。这是 React Native 开发中非常基础的组件。它们的 UI 特性基本上已经定型了，这两个组件在两个平台上的 UI 特性有一些区别，因此本章会详细讨论它们的 UI 特性，让读者能更快、更轻松地在开发中使用它们。

组件的引用在 React Native 开发中也是需要用到的技术。本章从 TextInput 组件的某些问题引出组件的引用这个概念，讨论它的特性以及如何使用组件的引用来解决问题。

6.1 Text 组件

Text 组件用来显示一段字符串。在 React Native 开发中，所有需要显示的字符串都需要放置在 Text 组件或者由它派生出的 TextInput 组件中。

6.1.1 样式键设置

Text 组件支持 View 的所有样式键设置。但是 Text 组件在布局上不同于其他组件：Text 内部的元素不再使用 flexbox 布局，而是采用文本布局。这意味着 Text 组件内部的元素不再是一个个矩形，当组件内部的元素将要排列出组件末端时会自动折叠添加新行。

基于这个特性，开发者可以只设定 Text 组件样式的宽度，而不设置它的高度，Text 组件的高度将由 Text 组件的宽度、需要显示的字符串长度、字体大小共同来动态确定。字符串长度与字体大小确定后，这个字符串的长度就可以计算出来，此时如果 Text 组件的宽度不够显示字符串的话，字符串将自动被分拆为多行显示直到全部显示出来。这时 Text 组件的高度就取决于所显示的字符串需要被分拆为多少行。采用这种不设置 Text 组件高度的办法，当 Text 组件显示的字符串动态改变时，Text 组件的高度也将随之动态改变。

假设 Text 组件中显示的字符串只有一行，并且我们设置了 Text 组件的高度，这个高度比字体的高度大很多，字符串将会贴着 Text 组件的上沿显示，开发者无法将它调整到垂直方向居中显示。

开发者不要将 Text 组件视为一个支持 flexbox 样式的盒子模型，不要按照盒子模型对其设置 flexbox 键值。在 3.5.1 节中的代码 3-6 的第 17~21 行，笔者使用了在显示字符串前加回车这个技巧，试了好几次对应样式键的值，才勉强做到了固定长度的字符串在 Text 组件中垂直方向居

中显示。如果 Text 组件要显示的字符串是可变的,那么想要通过 flexbox 设置让它居中显示是不可能的。

常见的做法是,使用一个 View 组件作为 Text 组件的容器,然后对这个 View 组件设置 flexbox 样式,从而达到我们希望的效果。本章 6.1.5 节将详细讨论这个技巧。

6.1.1.1 字体相关样式键设置

fontFamily 用来指定 Text 组件以何种字体族显示。它的取值有:sans-serif、serif、monospace,以及延伸出来的 sans-serif-light、sans-serif-thin、sans-serif-condensed、sans-serif-medium。这个样式键决定了英文字符串显示时的字体风格。

提示: fontFamily 是 UI 相关的概念,在这里简单介绍一下。在罗马字母阵营中,字体分为两大种类:Sans Serif 和 Serif。后来因为打字机的出现,又独立出 Monospace 这种等宽字体种类。Serif 字体族在字的笔画开始及结束的地方有额外的装饰,而且笔画的粗细会因直横的不同而有所不同。相反,Sans Serif 字体族没有额外的装饰,笔画粗细大致差不多。

fontStyle 用来指定字体的风格。它有两个字符型的取值:normal 和 italic,分别代表正常的字体和斜体字体。这个键值也只与英语字符显示有关。

fontSize 是数值类型的样式键,用来指定 Text 组件显示文字的大小。本章 6.2 节将详细讨论这个样式键的设置。

fontWeight 是字符串类型的样式键,它决定了字的粗细。它的取值有:normal、bold、100、200、300、400、500、600、700、800、900。其中,normal 和 bold 表示平时常说的正常字体与粗体;其后的 9 个数字序列代表从最细(100)到最粗(900)的字体粗细程度,每一个数字定义的粗细都要比上一个等级稍微粗一些。

6.1.1.2 其他样式键设置

textAlign 是字符串类型的样式键,它的取值有:auto、left、right、center、justify。当取值为 auto 时,将根据 Text 组件显示的字符语言来决定字符串如何排列,比如英语将自动从左向右排列,而阿拉伯语将自动从右向左排列。其他值的含义与 flexbox 中的同名值的含义相同。

letterSpacing 是数值型的样式键,用来指定字符串的每个字符之间插入多少空间。

writingDirection 是字符串类型的样式键,用来指定文本的书写方向。可以取值为:auto、ltr、rtl,分别表示自动(由字符语言决定)、从左到右和从右到左。

lineHeight 是数值类型的样式键,用来定义每一行的高度。

textDecorationLine 是字符串类型的样式键,取值为:none、underline、line-through、underline line-through,分别对应没有装饰线、下画线装饰、装饰线贯穿装饰、下画线贯穿装饰。

`textDecorationStyle` 是字符串类型的样式键，取值为：`solid`、`double`、`dotted`、`dashed`，分别对应实线装饰风格、双实线装饰风格、点状线装饰风格和虚线装饰风格。

`textDecorationColor` 是字符串类型的样式键，取值同其他的 `color`。

`textShadowOffset`、`textShadowRadius` 和 `textShadowColor` 三个键与阴影效果有关，我们在 6.1.4 节中讨论它们。

6.1.2 其他属性

`allowFontScaling`，布尔类型，iOS 操作系统专用，表示显示的字体是否要根据为失能者的设置而改变。

`numberOfLines`，数值类型，表示 `Text` 组件中的字符串可以显示为多少行。如果不设置的话，将按照字符串的长度显示，无行数限制直到显示完为止。当有限制时，新的行会将最上一行顶出显示区域。

`onLayout` 与前面讨论的 `onLayout` 行为类似。

`onPress` 不推荐使用，因为没有触摸的视觉反馈效果。

6.1.3 Text 组件的嵌套

开发者可以在开发中使用嵌套的 `Text` 组件。在嵌套的 `Text` 组件中，子 `Text` 组件将继承它的父 `Text` 组件的样式。当使用嵌套的 `Text` 组件时，子 `Text` 组件不能覆盖从父 `Text` 组件继承而来的样式，只能增加父 `Text` 组件没有指定的样式。如果试图在代码中覆盖从父 `Text` 组件继承而来的样式，覆盖将不会生效，并且在开发模式下会弹出警告。示例代码见代码 6-1。注意，在代码 6-1 中，在嵌套 `Text` 组件的显示字符串中，希望重起一行显示的字符串必须要在字符串前加 `{'\r\n'}`，否则会接着上一行末显示。

代码 6-1:

```
.....
render: function() {
  return (
    <View style={styles.container}>
      <Text style={{fontSize:20,textAlign:'center'}}>
        我是 20 号字体
        <Text style={{fontWeight: 'bold'}}>
          {'\r\n'}我是加粗 20 号字体
          <Text style={{color: 'black'}}>
            {'\r\n'}我是加粗黑色 20 号字体
          </Text>
        </Text>
      </Text>
    </View>
  );
}
.....
```

这段代码在手机屏幕上的显示效果（仅截取显示部分）如图 6-1 所示。



图 6-1 代码 6-1 运行效果

6.1.4 文本显示的阴影效果

从 React Native 0.18.0 版本开始，Text 组件开始支持阴影效果。为了实现阴影效果，需要设定三个样式键：textShadowOffset、textShadowRadius 和 textShadowColor。它们的使用参见代码 6-2。

代码 6-2:

```
.....
render() {
  let baseStyle = {
    fontSize:20, textAlign:'center', color:'black',
    textShadowOffset: {width: 5, height: 5},
    textShadowRadius: 2, textShadowColor: 'grey' //定义阴影效果相关样式键
  };
  return (
    <View style={styles.container}>
      <Text style={baseStyle}>
        我是 20 号字体
        <Text style={{fontWeight: 'bold'}}>
          {'\r\n'}我是加粗 20 号字体
        </Text>
      </Text>
    </View>
  );
}
.....
```

代码 6-2 运行效果参见图 6-2。

注意，第二排的阴影效果没有显示完全，被 Text 组件自动生成的宽和高在右侧与下侧各切了一小部分。解决这个问题很简单，在 baseStyle 样式中增加：padding:5。padding 的取值与 textShadowOffset 中的最大值相等。为了美观，我们再加入一个样式：letterSpacing:5。修改后的显示效果见图 6-3。

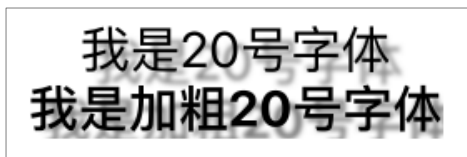


图 6-2 代码 6-2 运行效果

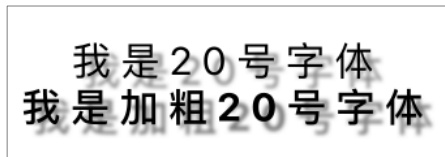


图 6-3 修改代码 6-2 后运行效果

6.1.5 Text 居中显示

在开发中，开发者有时需要将一串字符串显示在比字符串高度高很多的空间中，这时通常的需求就是让字符串居中（水平与垂直方向都居中）显示在一个方形区域中。

在 flexbox 样式中定义了如何水平居中和垂直居中显示，把它们用在 Text 组件的样式中。实现如代码 6-3 所示。

代码 6-3:

```
'use strict';
var React = require('react-native');
var {
  AppRegistry, StyleSheet, Text, View
} = React;
var Project19 = React.createClass({
  render: function() {
    return (
      <View style={styles.container}>
        <Text style={styles.textStyle}>
          happy
        </Text>
        <Text style={styles.textStyle}>
          忧伤
        </Text>
      </View>
    );
  }
});
var styles = StyleSheet.create({
  container: {
    flex:1,
    justifyContent: 'center',
    alignItems: 'center'
  },
  textStyle: {
    height:100,
    width:200,
    fontSize: 30,
    backgroundColor: 'gray',
    textAlign: 'center',
    justifyContent: 'center',    //虽然样式中设置了 justifyContent: 'center', 但无效
    margin:5
  }
});
AppRegistry.registerComponent('Project19', () => Project19);
```

在这段代码中，Text 组件的样式键 `textAlign` 和 `justifyContent` 都设置为 `center`，这样字符串应当垂直和水平方向都居中显示了吧。事实上，只会水平居中显示，这段代码在两个平台上运行的效果都如图 6-4 所示。



图 6-4 代码 6-3 运行效果

正确的做法如代码 6-4 所示。

代码 6-4:

```
'use strict';
var React = require('react-native');
var {
  AppRegistry, StyleSheet, Text, View,
} = React;
var Project19 = React.createClass({
  render: function() {
    return (
      <View style={styles.container}>
        <View style={styles.viewForTextStyle} > //在 Text 外加一个包围的 View
          <Text style={styles.textStyle}>
            happy
          </Text>
        </View>
        <View style={styles.viewForTextStyle} > //在 Text 外加一个包围的 View
          <Text style={styles.textStyle}>
            忧伤
          </Text>
        </View>
      </View>
    );
  }
});
var styles = StyleSheet.create({
  container: {
    flex:1,
    justifyContent: 'center',
    alignItems: 'center'
  },
  textStyle: {
    fontSize: 30 //除了 fontSize 样式键，其他的样式键都移动到包围 Text 组件
  }, //的 View 组件样式中
  viewForTextStyle: { //包围 Text 组件的 View 组件的样式设置
    height:100,
```



```

    width:200,
    alignItems:'center',
    justifyContent: 'center',
    backgroundColor: 'gray',
    margin:5
  }
});
AppRegistry.registerComponent('Project19', () => Project19);

```

这段代码在两个平台上运行的效果都如图 6-5 所示，达到了开发者期望的字符串在指定范围内水平和垂直方向都居中显示的要求。



图 6-5 代码 6-4 运行效果

6.1.6 在字符串中插入图像

在 React Native 0.20.0 版本中，开发者可以在 Text 组件中更方便地插入图像。示例代码参见代码 6-5。

代码 6-5, index.android.js 或者 index.ios.js:

```

'use strict';
var React = require('react-native');
var {
  AppRegistry, StyleSheet, Text, View, Image
} = React;
var aImage = require('./1.jpg');

var Project20 = React.createClass({
  render: function() {
    return (
      <View style={styles.container}>
        <Text style={styles.welcome}>
          Welcome to <Image source={aImage} style={styles.imageInTextStyle}/>
        </Text>
      </View>
    );
  }
});

```

```
var styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#F5FCFF',
  },
  welcome: {
    fontSize: 20,
    textAlign: 'center',
    margin: 10,
  },
  imageInTextStyle: {
    width: 30,
    height: 30,
    resizeMode: 'cover'
  }
});
AppRegistry.registerComponent('Project20', () => Project20);
```

代码 6-5 运行效果参见图 6-6。

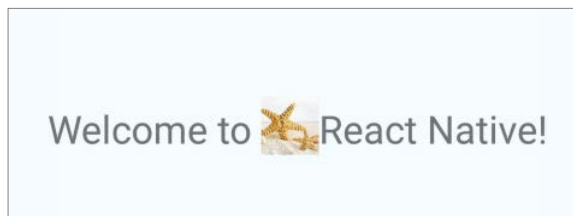


图 6-6 代码 6-5 运行效果

6.2 Text 组件在两个平台上的不同表现

开发者在使用 Text 组件时，比较麻烦的就是在一些情况下，Text 组件在两个平台上的表现不一样。现在分样式定义的不同情况讨论如下。

代码 6-6 是本节讨论 Text 组件在两个平台上的不同表现时所用的代码，截图也是这段代码按情况稍加修改后在手机上运行的效果。

代码 6-6，index.android.js 或者 index.ios.js:

```
'use strict';
var React = require('react-native');
var {
  AppRegistry,
  StyleSheet,
  Text,
  View,
} = React;
var Project19 = React.createClass({
  render: function() {
    return (
      <View style={styles.container}>
```

```

        <Text style={styles.welcome}>
            Ajfg 你好
        </Text>
    </View>
    );
}
});
var styles = StyleSheet.create({
    container: {
        flex:1,
        justifyContent: 'center',
        alignItems: 'center'
    },
    welcome: {                //Text 组件的样式声明
        width:200,
        height:100,
        fontSize:50,
        backgroundColor: 'grey'
    }
});
AppRegistry.registerComponent('Project19', () => Project19);

```

6.2.1 只指定 fontSize，不指定 height

在这种情况下，Text 组件在两个平台上显示都正常，效果如图 6-7 所示。

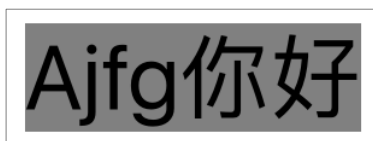


图 6-7 Text 组件显示效果 1

提示：为什么显示 jfg? 因为在小写字母中，f 最高，g 最低，j 又高又低。

在图 6-7 中，灰色区域代表 Text 组件的宽和高，因为水平方向容易调整，所以这里只讨论垂直方向。可以看到，在垂直方向上，Text 组件要比字高，上下都留有富余的空间，这样显示出来美观。但如果仔细看，就会发现 Android 平台的显示下方所留的空间比 iOS 平台的显示下方所留的空间要小一些。或者这么说，图 6-7 是 iOS 平台效果图，两个汉字上、下方富余的空间基本相等；而在 Android 平台上，上方富余空间的高度大概是下方空间的 1.5 倍。除非字体比较大，否则用户不容易发现。

6.2.2 只指定 height，不指定 fontSize

在这种极端情况下，不论 height 是何值，fontSize 的值都是 13。

6.2.3 fontSize 等于 height

在这种情况下，iOS 平台与 Android 平台的表现不同。在 iOS 平台的显示效果如图 6-8 所示。

可以看到，字的上方还有空间，但字的最下一部分没有显示出来。在 Android 平台上，这种情况更严重。Android 平台的显示效果如图 6-9 所示。

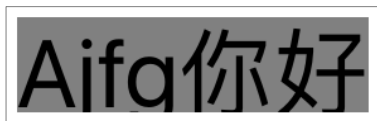


图 6-8 Text 组件显示效果 2



图 6-9 Text 组件显示效果 3

这两种显示情况，即使在 Text 组件样式中加入 `padding:0` 样式键或者 `paddingTop:0` 和 `paddingBottom: 0` 两个样式键，也仍然不会有任何改变。

6.2.4 height 大于 fontSize

在 iOS 平台上，当 `height` 等于 `fontSize` 的 1.2 倍时，显示效果与只指定 `fontSize`、不指定 `height` 类似。如果 `height` 继续增大，此时 Text 组件中显示字符的上方空间保持不变，而下方空间会随着 `height` 的增大而增大。

在 Android 平台上，当 `height` 等于 `fontSize` 的 1.35 倍时，显示效果与只指定 `fontSize`、不指定 `height` 类似。如果 `height` 继续增大，此时 Text 组件中显示字符的上方空间保持不变，而下方空间会随着 `height` 的增大而增大。

这种情况的效果如图 6-10 所示。



图 6-10 Text 组件显示效果 4

6.2.5 边框在两个平台上的不同表现

因为 Text 组件继承了 View 组件的样式设置，所以开发者认为可以使用 `borderWidth` 给 Text 组件设置一个边框。代码 6-7 取消了原来 Text 组件的背景色为灰色的设置，并设置边框宽度为 1。

代码 6-7:

```
.....
var styles = StyleSheet.create({
  container: {
    flex:1,
    justifyContent: 'center',
    alignItems: 'center'
```

```

    },
    welcome: {
      width:200,
      fontSize:50,
      borderWidth:1      //设置边框宽度为 1
    }
  },
  .....

```

在 iOS 平台上，修改后的代码运行效果如图 6-11 所示。

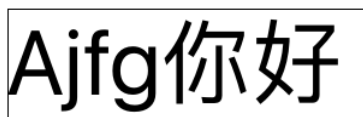


图 6-11 Text 组件显示效果 5

但在 Android 平台上运行相同的代码时，边框不会出现。效果图在此不给出了，大家把图 6-11 去掉字符串四周的边框，就是代码在 Android 平台上运行的效果。

为了实现在 Android 平台上 Text 组件能有边框，开发者需要用一个 View 组件包围住 Text 组件。这个 View 组件只需要有一个样式键 `borderWidth`。当然，如果开发者需要，也可以再加入其他样式键与属性。代码 6-8 示范了如何在 Android 平台上实现带边框的 Text 组件。

代码 6-8:

```

'use strict';
var React = require('react-native');
var {
  AppRegistry,
  StyleSheet,
  Text,
  View,
} = React;
var Project19 = React.createClass({
  render: function() {
    return (
      <View style={styles.container}>
        <View style={{borderWidth:1}}>      //为了实现边框而加入的 View
          <Text style={styles.welcome}>
            Ajfg 你好
          </Text>
        </View>
      </View>
    );
  }
});
var styles = StyleSheet.create({
  container: {
    flex:1,
    justifyContent: 'center',
    alignItems: 'center'
  },
  welcome: {
    width:200,      //在 Text 组件的样式声明中不需要有 borderWidth
                  //如果有 borderWidth，它也不会起任何作用
  }
});

```

```

        fontSize:50
      }
    });
    AppRegistry.registerComponent('Project19', () => Project19);

```

代码 6-8 在 Android 平台与 iOS 平台上运行的效果如图 6-11 所示。

当开发者需要实现带边框的 Text 组件时，建议统一使用代码 6-8 的方式，这种方式在两个平台上都能实现同样的效果。

6.3 TextInput 组件

React Native 框架向开发者提供了 TextInput 基础组件，用户可以在这个组件中通过键盘输入文字。

6.3.1 TextInput 组件样式键

TextInput 组件可以使用 View 组件和 Text 组件的所有样式键，它没有自己特有的样式键。

与 Text 组件类似，TextInput 组件内部的元素不再使用 flexbox 布局，而是采用文本布局。这意味着 TextInput 组件内部的元素不再是一个个矩形，当组件内部的元素将要排列出组件末端时会自动折叠添加新行。请开发者不要将 TextInput 组件视为一个支持 flexbox 样式的盒子模型，不要按照盒子模型对其设置 flexbox 键值。

6.3.2 TextInput 组件的属性

TextInput 组件的 onLayout 回调函数的使用方法与 View 组件的 onLayout 回调函数类似。TextInput 组件的其他回调函数类型的属性将在 6.5 节中讨论。

autoCapitalize 是字符串类型的属性，它的取值可以为：none、sentences、words、characters，分别表示不自动变为大写、将每句话的首字母自动改为大写、将每个单词的首字母自动改为大写、将每个英文字母自动改为大写。

autoCorrect 是布尔类型的属性，用来定义 TextInput 组件是否自动更正用户的输入。它有一个默认值 true，会默认自动更正用户的输入。自动更正用于检查用户输入的英语单词是否正确。

autoFocus 是布尔类型的属性，用来定义 TextInput 组件是否自动获得焦点。它的默认值是 false。

defaultValue 是字符串类型的属性，用来定义 TextInput 组件中的字符串默认值。

editable 是布尔类型的属性，用来定义 TextInput 组件是否允许用户修改组件内的字符。当其值为 false 时，不允许修改。

keyboardType 是字符串类型的属性。它的取值为：default、numeric、email-address、ascii-capable、numbers-and-punctuation、url、number-pad、phone-pad、name-phone-pad、decimal-pad、twitter、web-search。它定义了当 TextInput 组件获得焦点时，将自动弹出哪种软键盘。default、numeric、

email-address 这三种是 Android 平台和 iOS 平台都支持的键盘类型。

MaxLength 是数值类型的属性，用来定义 TextInput 组件最多允许用户输入多少个字符。

multiline 是布尔类型的属性。当它为真时，TextInput 组件可以是多行的组件。它的默认值是 false。

placeholder 是字符串类型的属性。它在 TextInput 组件被渲染时会显示在组件内，但颜色比正常输入的字符淡些，并且用户在 TextInput 组件中输入第一个字符时，它就会消失。

placeholderTextColor 定义了 placeholder 字符串的颜色。

secureTextEntry 和 password 都是布尔类型的属性。它们中只需要定义一个即可。它们定义了当前 TextInput 组件是否用于输入密码。其效果在第 2 章中读者已经看到。

value 是字符串类型的属性，用来设置 TextInput 组件内字符串的值。要慎重使用这个属性，因为它有可能会带来屏幕显示闪烁。React Native 官方更推荐使用 editable 属性和 defaultValue 属性来达到相同的效果。但当应用需要突然改变 TextInput 组件内字符串的值时，还是需要使用这个属性。6.7.4 节中的代码 6-14 示范了如何使用这个属性。

onSelectionChange 是回调函数类型的属性。当用户在文本输入框中选择的字符串发生改变时，这个回调函数将被调用，并且会被传入一个 event 参数。这个 event 参数对开发者有用的数据结构如下：

```
{
  timeStamp: 事件发生时的时间值
  nativeEvent: {
    selection: {
      start: 用户选中的子字符串起点位置
      end: 用户选中的子字符串结束位置
    }
  }
}
```

从 event 中开发者可以得到用户在输入框中选择一段字符串这个事件发生的时间，还可以得到用户选择的子字符串在输入框中的起点位置与结束位置。

作为一个特例，当用户移动输入光标时，这个事件也会被触发（如果开发者设置了这个回调函数的话）。比如：用户把 TextInput 组件中的输入光标移动到最开始的位置，这时 event.nativeEvent.selection.start 和 event.nativeEvent.selection.end 的值都是 0。其余情况依此类推。

6.3.3 TextInput 组件 iOS 平台专有属性

blurOnSubmit 是布尔类型的属性。当它为真时，输入完成提交时，文本区域会被模糊化。对于单行的 TextInput 组件，它的默认值是 true；而对于多行的 TextInput 组件，它的默认值是 false。请注意，对于多行的 TextInput 组件，将 blurOnSubmit 设为 true 会使 multiline 属性的值失效，TextInput 组件成为一个单行输入的组件。用户按下键盘上的回车键时会模糊化输入的文本并触发

onSubmitEditing 事件，而不是在输入区域插入新行。

clearButtonMode 是字符串类型的属性。它的取值为：never、while-editing、unless-editing、always。它定义了什么时候在文本 View 的右侧显示清除按钮。

clearTextOnFocus 是布尔类型的属性。当它为真时，当用户点击 TextInput 组件开始编辑文字时，会自动清除文字区域。

enablesReturnKeyAutomatically 是布尔类型的属性。它的默认值为 false。当它为真时，在文本区域没有输入文字时，键盘的回车键会失效；而有文字时，键盘的回车键又会生效。

keyboardAppearance 是字符串类型的属性。它的取值为：default、light、dark，分别表示默认、明亮、偏暗三种不同的键盘颜色。

onKeyPress 是一个回调函数。当 TextInput 组件获得焦点后，一个按键被按下时，这个回调函数将被调用并被传入按下键的键值。这个函数会在 onChange 回调函数之前被调用。

returnKeyType 是字符串类型的属性。它的取值为：default、go、google、join、next、route、search、send、yahoo、done、emergency-call。当 TextInput 组件获得焦点时，它定义了回车键在键盘布局中的外表。

selectTextOnFocus 是布尔类型的属性。当它为真时，如果 TextInput 组件获得焦点，组件中所有的文字都会被选中。

tintColor 是颜色类型的属性，使得 TextInput 组件输入的字符串有染色效果。

6.3.4 TextInput 组件 Android 平台专有属性

numberOfLines 是数值类型的属性，用来设置 TextInput 组件可以有多少行。将它与 multiline={true} 联合使用，可以让用户输入多行。

textAlign 是字符串类型的属性。它的取值为：start、center、end。它定义了文字在 TextInput 组件中显示的位置。

textAlignVertical 是字符串类型的属性。它的取值为：top、center、bottom。它定义了 TextInput 组件中的文字在垂直方向上的位置。

underlineColorAndroid 是字符串类型的属性，用来定义输入提示下画线的颜色。如果将它的颜色设为与 TextInput 组件的背景色一样，则可以隐藏输入提示下画线。

6.3.5 TextInput 组件的成员函数

TextInput 组件有一个名为 focus 的成员函数。当得到一个 TextInput 组件的引用（如何获得组件的引用，将在 6.7 节中详细讨论）后，假设这个 TextInput 组件的引用名为 aTextInputRef，则可以使用：

```
this.refs.aTextInputRef.focus();
```


语句让这个 TextInput 组件获得焦点。

假设开发者有一个界面，界面中有多个 TextInput 组件，那么当一个 TextInput 组件失去焦点时，可以对用户的输入进行检查，如果用户的输入非法，则可以显示用户的输入为什么非法，同时使用 focus 函数让焦点再次回到刚失去焦点的输入框，以便用户重新输入。

focus 函数还可用于让焦点从一个输入框跳到下一个输入框。这些内容待读者学习完本章 TextInput 的所有知识后，都可以自己设计、实现业务逻辑。

6.4 TextInput 组件在两个平台上的不同表现

TextInput 组件在两个平台上的表现同样也有所不同。代码 6-9 是本节讨论 TextInput 组件在两个平台上显示不同时的基准代码，请读者在此基础上对相应的样式属性进行修改。

代码 6-9:

```
'use strict';
var React = require('react-native');
var {
  AppRegistry, StyleSheet, TextInput, View,
} = React;
var Project19 = React.createClass({
  render: function() {
    return (
      <View style={styles.container}>
        <TextInput style={styles.textInputStyle}
          defaultValue='Ajfg 你好' />
      </View>
    );
  }
});
var styles = StyleSheet.create({
  container: {
    flex:1,
    justifyContent: 'center',
    alignItems: 'center'
    backgroundColor: 'white'
  },
  textInputStyle: {
    width:200,
    height:70,
    fontSize:50,
    alignItems:'center',
    justifyContent: 'center'
  }
});
AppRegistry.registerComponent('Project19', () => Project19);
```

6.4.1 Android 平台的输入下画线

代码 6-9 在 Android 平台上运行的效果如图 6-12 所示。在输入框区域有一条下画线，提醒用

户这里是一个输入框。但在 iOS 平台上没有这条下画线。

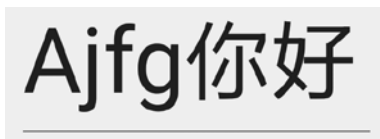


图 6-12 代码 6-9 运行效果（Android 平台）

这条下画线默认的颜色值是'grey'。开发者通过让这条下画线的颜色与 `TextInput` 组件的背景色相同，或者让 `TextInput` 组件的背景色比下画线的颜色略深，就可以达到让这条下画线消失的视觉效果。

在本例中，我们为 `TextInput` 组件的样式增加一个背景色 `grey` 来达到让下画线消失的视觉效果。将 `TextInput` 组件的背景色设为灰色，也可以让我们更清楚地看到 `TextInput` 组件的显示区域。

6.4.2 父组件的 `alignItems` 键失效

当开发者在一个组件中声明 `alignItems: 'center'` 时，期望这个组件中的子组件都会居中对齐（效果见第 5 章 5.1.2.4 节中的图 5-4）。在代码 6-9（修改背景色后）中的根 `View` 组件样式中声明了 `alignItems: 'center'`。

在图 6-13 中，左图是代码 6-9 在 Android 平台上运行的效果，右图是在 iOS 平台上运行的效果。可以看到，在 Android 平台上，`TextInput` 组件的父组件样式键 `alignItems: 'center'` 工作正常，`TextInput` 组件被居中对齐；而在 iOS 平台上，父组件样式键 `alignItems: 'center'` 没有发挥作用，`TextInput` 组件的对齐规则是从最左侧对齐。

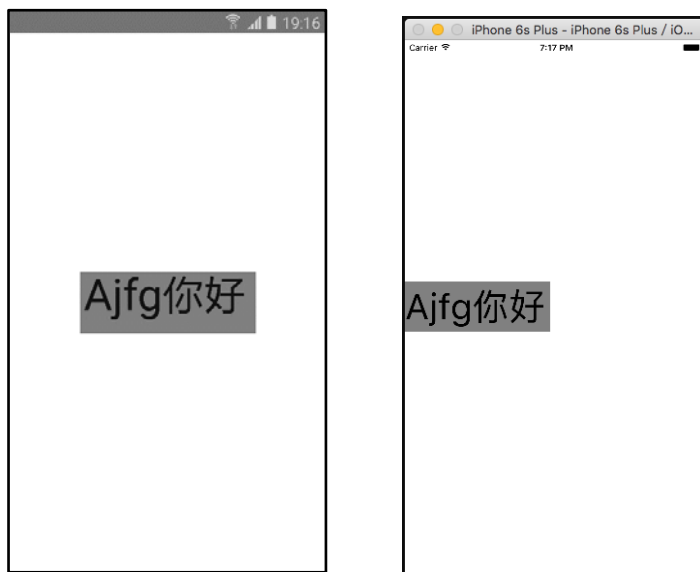


图 6-13 代码 6-9 在两个平台上的不同显示效果

为了让 iOS 平台的 `TextInput` 组件遵从父组件的对齐规则，需要将 `TextInput` 组件使用一个没

有任何属性的 View 组件包围起来，如代码 6-10 所示。

代码 6-10:

```
.....
var Project19 = React.createClass({
  render: function() {
    return (
      <View style={styles.container}>
        <View> //添加的不带任何属性的 View 组件
          <TextInput style={styles.textInputStyle}
            defaultValue='Ajfg 你好' />
        </View> //添加的不带任何属性的 View 组件结束
      </View>
    );
  }
});
.....
```

修改后，在 iOS 平台上运行代码 6-10，显示效果如图 6-13 左图所示，TextInput 组件达到了居中对齐的效果。

6.4.3 只指定 fontSize、不指定 height

在 iOS 平台上，如果 TextInput 组件只指定了 fontSize，没有指定 height，这时 height 会取默认值 0。也就是说，在 iOS 平台上，没有指定样式中 height 键值的 TextInput 组件不会显示。

在 Android 平台上，如果 TextInput 组件样式中只指定了 fontSize，没有指定 height，这时 height 会根据 fontSize 动态调整，保证输入字符串的上、下方都有比较多的富余空间。TextInput 组件的显示效果如图 6-14 所示。



图 6-14 TextInput 组件显示效果 1

6.4.4 height 等于 fontSize

在 Android 平台上，当 height 等于 fontsize 时，显示效果如图 6-15 所示。可以看到 TextInput 组件中的字明显被遮盖。

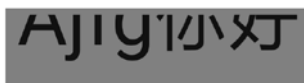


图 6-15 TextInput 组件显示效果 2

在 iOS 平台上，当 height 等于 fontsize 时，显示效果如图 6-16 所示。可以看到字的下方被略

为遮盖一点儿。

在 `TextInput` 组件的样式中再加入样式键：

```
paddingTop:0,
paddingBottom: 0
```

然后在 Android 平台上运行代码，显示效果如图 6-17 所示。



图 6-16 TextInput 组件显示效果 3



图 6-17 TextInput 组件显示效果 4

6.4.5 height 大于 fontSize

在 Android 平台上，当指定了上、下 padding 为 0 时，从 height 大于 fontSize 的 1.1 倍起，可以认为输入的字符串会始终处于 `TextInput` 组件区域的中部。

在 iOS 平台上，不论是否指定了上、下 padding 为 0，从 height 大于 fontSize 的 1.1 倍起，可以认为输入的字符串会始终处于 `TextInput` 组件区域的中部。

6.4.6 边框在两个平台上的不同表现

有时开发者希望 `TextInput` 组件的底色与外界相同，这时就需要使用边框来突出 `TextInput` 组件所在区域是一个输入框。因为 `TextInput` 组件的样式支持 flexbox 样式，那么只要在 `TextInput` 组件的样式中加入 `borderWidth:1`，就应当可以实现给 `TextInput` 组件增加边框了。请读者在代码 6-10 的基础上，在 `TextInput` 组件的样式中加入 `borderWidth:1`，删除 `backgroundColor` 键，然后运行修改后的代码。

运行后，读者可以看到 iOS 平台的 `TextInput` 组件正确显示出了边框，而 Android 平台的 `TextInput` 组件没有显示出边框。让 Android 平台的 `TextInput` 组件正确显示边框的方法与 `Text` 类似，把 `TextInput` 组件用一个 `View` 组件包围起来。实现代码见代码 6-11。

代码 6-11：

```
.....
render: function() {
  return (
    <View style={styles.container}>
      <View style={{borderWidth:1}}>          //用一个边框宽为 1 的 View 组件
        <TextInput style={styles.textInputStyle}    //包围 TextInput
          defaultValue='Ajfg 你好'
          underlineColorAndroid='white' />        //设置 TextInput 组件下划线颜色
      </View>
    </View>
  );
}
.....
```

代码 6-11 在两个平台上运行时手机 UI 界面显示都类似于图 6-18 所示的效果。

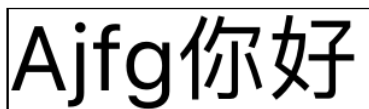


图 6-18 TextInput 组件显示效果 5

6.5 TextInput 组件的生命周期

一个被加载的 TextInput 组件可以由它的各个回调事件构成生命周期。

6.5.1 获得焦点

当用户点击输入框时，TextInput 组件获得焦点，如果指定了 onFocus 属性，则相应的回调函数将被调用。

回调函数将得到一个 event 参数。如果 TextInput 组件是单行的组件，并且应用运行在 iOS 平台上，开发者可以通过 event.nativeEvent.text 获得组件中的字符串（是用户上次输入的，或者是程序设定的默认值，或者是程序设定的值）；否则 event.nativeEvent.text 的值是 undefined。

6.5.2 用户输入

用户在获得焦点的输入框中的每一次输入（添加一个字符或者删除一个字符），onChangeText 和 onChange 回调函数都会被调用。

onChangeText 回调函数已经在第 2 章讨论过，这里不再重述。

onChange 回调函数与 onChangeText 类似，但它接收的参数是一个 event。在这个 event 中，对普通开发者唯一有用的就是用户输入的字符串，可以通过 event.nativeEvent.text 得到。更简单的办法是使用 onChangeText 回调函数，而不去处理 onChange 事件。

6.5.3 用户按下提交键

当用户按下提交键后，如果指定了 onSubmitEditing 属性对应的回调函数，则该函数会被调用。回调函数将得到一个 event 参数，开发者可以通过 event.nativeEvent.text 得到用户输入的字符串。

提示：什么是提交键？这取决于开发者给当前 TextInput 组件指定的软键盘类型，提交键可以是回车键、提交键和发送键中的一个。

当开发者通过 multiline={true} 语句将一个 TextInput 组件设为多行的 TextInput 组件时：

- 在 iPhone 手机上，这个 TextInput 组件的 onSubmitEditing 事件永远也不会被触发，onSubmitEditing 回调函数也永远不会被执行（如果开发者提供了这个回调函数）。
- 在 Android 手机上，用户每一次按下回车键后，onSubmitEditing 事件都会被触发，但输

入框中的字符串会增加一个回车换行，同时输入框会继续保持住焦点，不会进入失去焦点生命周期中。

如果 `TextInput` 组件是单行的，`onSubmitEditing` 回调函数执行完成后，会进入 6.5.4 节描述的失去焦点生命周期中。

6.5.4 失去焦点

在两种情况下拥有焦点的 `TextInput` 组件会失去焦点：

- 在单行的输入框中用户按下提交键；
- 用户直接点击了另一个 `TextInput` 组件。

请读者注意只有这两种情况 `TextInput` 组件才会失去焦点。此时失去焦点后续事件才会被触发。

当一个输入框失去焦点时，React Native 框架会依次调用 `TextInput` 组件的 `onEndEditing` 和 `onBlur` 回调函数。

这两个函数都会得到一个 `event` 参数，开发者可以通过 `event.nativeEvent.text` 得到用户输入的字符串。但 `onBlur` 得到的 `event` 在 React Native 0.19.0 版本中还有问题，对多行的 `TextInput` 取不出用户输入的字符串。因此如果要靠 `TextInput` 组件失去焦点事件取得用户输入的字符串，应当选择 `onEndEditing` 回调函数。

如果用户在一个输入框中输入了字符串后，点击界面上的开发者设置的按钮来提交输入，这时失去焦点事件不会发生，因此开发者不能通过 `onEndEditing` 或者 `onBlur` 回调函数得到用户最终输入的字符串。这也是为什么通常都使用 `onChangeText` 事件来时刻获取用户输入的原因。

6.6 软键盘与键盘事件

键盘与 `TextInput` 组件是一个困扰了 React Native 开发者比较久的问题。对这个问题描述如下：

假设在手机屏幕的底部有一个 `TextInput` 输入框，当用户点击这个输入框时，手机的软键盘会自动弹出，以供用户进行输入。

在 React Native 0.18.0 版本之前，软键盘会遮盖住 `TextInput` 组件（可能还有些其他的组件），导致用户可以输入，但看不到自己输入了什么……React Native 开发者社区想出了很多方法来解决这个问题。

React Native 0.18.0 版本解决了一半这个问题。在 Android 手机上，`TextInput` 组件及其上方的组件都会随着软键盘的弹出而自动上移（有部分会被顶出屏幕），这样用户可以边输入边看到自己输入了什么，并且当用户让软键盘消失时所有被上移的组件的显示位置都会复位。但在 iPhone 手机上，软键盘还是会毫不客气地遮盖住 `TextInput` 组件。

本书出版时，React Native 大概会发布到 0.22.0 版本，也许这个问题已经得到全部解决。但这个问题是一个比较好的讨论 React Native 开发对键盘事件处理的机会，因此本书在此借这个问题来

讨论键盘事件与事件处理。

为了解决软键盘可能遮住 TextInput 组件的问题，开发者需要对键盘事件进行监听，能够检测到键盘弹出与键盘收回，能够在应用需要时通过代码关闭键盘。

代码 6-12 在手机屏幕的最下方放置了一个 TextInput 组件，然后在上方放置了一个按钮用来手动关闭键盘。

代码 6-12:

```
'use strict';
let React = require('react-native');
let Dimensions = require('Dimensions');
let totalHeight = Dimensions.get('window').height;
var {
  AppRegistry, StyleSheet, Text, View,
  DeviceEventEmitter, TextInput //导入 DeviceEventEmitter 监听事件
} = React;
var Project19 = React.createClass({
  getInitialState:function() {
    return {
      KeyboardShown:false //这个变量用来记录软键盘是否已经被弹出
    };
  },
  //定义键盘已经弹出事件监听器，不能使用箭头函数定义，因为卸载监听器时需要用到引用
  keyboardDidShowListener:function(event) {
    this.setState({KeyboardShown: true});
  },
  //定义键盘已经隐藏事件监听器，不能使用箭头函数定义，因为卸载监听器时需要用到引用
  keyboardDidHideListener:function(event) {
    this.setState({KeyboardShown: false});
  },
  componentWillMount:function() {
    //下面这条语句开始监听键盘已经弹出事件，并设置处理这个事件的函数

    DeviceEventEmitter.addListener('keyboardDidShow',this.keyboardDidShowListener);
    //下面这条语句开始监听键盘已经隐藏事件，并设置处理这个事件的函数

    DeviceEventEmitter.addListener('keyboardDidHide',this.keyboardDidHideListener);
  },
  componentWillUnmount:function() {
    //卸载两个事件监听器
    DeviceEventEmitter.removeListener('keyboardDidShow',this.keyboardDidShowListen);
    DeviceEventEmitter.removeListener('keyboardDidHide',this.keyboardDidHideListen);
  },
  onDismissKeyboard:function() { //这个函数用来强制隐藏键盘
    let dismissKeyboard = require('dismissKeyboard')
    dismissKeyboard();
  },
  render: function() {
    return (
      <View style={[styles.container, this.state.KeyboardShown &&
        styles.bumpedContainer]}> //按照键盘是否弹出设置窗口的样式，让 TextInput 组件避免被遮盖
        <Text style={styles.buttonStyle}
```

```
        onPress={this.onDismissKeyboard}>
          Dismiss Keyboard
        </Text>
        <TextInput style={styles.textInputStyles}
          onFocus={() => this.setState({bumpedUp: 1})}
          onEndEditing={() => this.setState({bumpedUp: 0})}/>
      </View>
    );
  }
});
var styles = StyleSheet.create({
  container: {
    flex: 1,
  },
  bumpedContainer: { //用来让父容器整体上移的样式
    marginBottom: 210,
    marginTop: -210,
  },
  buttonStyle: {
    top: 250,
    fontSize: 30,
    backgroundColor: 'grey'
  },
  textInputStyles: {
    position: 'absolute',
    top: totalHeight - 80,
    left: 20,
    width: 200,
    height: 30,
    fontSize: 20,
    backgroundColor: 'grey'
  }
});
AppRegistry.registerComponent('Project19', () => Project19);
```

代码 6-12 在 Android 与 iPhone 手机上执行有区别，下面进行一一讲解。

提示：iOS 平台还支持监听 keyboardWillHide 事件，但因为 Android 平台不支持这个事件，所以在例程中没有监听这个事件。

在 Android 平台上，键盘事件处理函数能收到一个 event 参数，可以在 event 参数中取到下列 4 个值：

```
event.endCoordinates.screenX
event.endCoordinates.screenY
event.endCoordinates.width
event.endCoordinates.height
```

在 iPhone 平台上，可以在 event 参数中取到下列 10 个值。

从接收到的键盘事件中，开发者可以取到：

```
event.easing: 这个值始终是 keyboard
event.duration: 记录软键盘弹出动画的持续时间，单位是毫秒
```


还可以取到下面 8 个变量，但直到 React Native 0.19.0 版本，这 8 个值都是 undefined。

```
event.startCoordinates.screenX
event.startCoordinates.screenY
event.startCoordinates.width
event.startCoordinates.height
event.endCoordinates.screenX
event.endCoordinates.screenY
event.endCoordinates.width
event.endCoordinates.height
```

这些值的含义从它们的英文名称就可以推断出来。在制作某些很精美的效果时，可能会用到这些值。这里不再详述。

6.7 组件的引用

在前几章的例程中，我们已经使用了许多 React Native 提供的组件，也使用了自定义组件。在这些组件中，我们都没有定义组件的标识。使用逻辑是：当某个组件出现需要处理的事件时，由这个组件自己上报该事件，然后对应的事件处理代码对事件进行处理。在大部分时候，这种机制都能满足开发者的需求。但在有些情况下，我们需要对组件进行操作，此时就需要能够得到组件的引用，进而通过该引用对组件进行操作。

6.7.1 定义组件引用

在某个组件的 JSX 代码描述中加入 `ref={某字符串}`，就可以定义一个组件的引用名称。例如：

```
.....
<TextInput ref='aReferName'
.....
```

在上面的这段代码中，我们描述了一个 `TextInput` 组件，它的引用名称为 `'aReferName'`。那么当我们需要时，就可以通过 `this.refs.aReferName` 得到这个组件的引用。

6.7.2 得到系统定义的组件引用

通常，组件代码都是由开发者编写的，在编写组件时如果需要使用到组件的引用，就会在组件的定义代码中给组件一个便于记忆的引用名称，就像 6.7.1 中的示例那样。

但还有一种特殊情况，在这种情况下，我们利用一个数组定义多个组件，组件的个数等于数组的长度。组件的定义是由代码完成的，开发者无法再一一对组件赋予引用名称。这时就需要在代码运行时取得系统定义的每个组件的引用，并将这些引用保存在一个数组中。

代码 6-13 示范了如何得到系统定义的组件引用。

代码 6-13：

```
.....
{ this.aArray.map( (aValue, aIndex) =>           //如果不熟悉数组的 map 函数，请参见附录 A.6
  <ComponentA aPropName={aValue}
```

```

      ref={ (refName) => { this.componentRefNames[aIndex] = refName; }}
      key={aOption}/> )
    }
    .....

```

在执行到代码 6-13 前，aArray 中已经存放好了数据。按照它的长度，代码 6-13 将生成多个 ComponentA（笔者虚拟的组件名）组件，并把每一个 ComponentA 组件的 aPropName 属性都被赋值为 aArray 中一个元素的值，然后每一个 ComponentA 组件的引用都存放在名为 componentRefName 的数组中以备后用。

提示：这一部分属于比较高级的 React Native 开发技术，初学者可能不容易理解，暂时心里有个印象即可，需要用到时再回头来看。

6.7.3 调用组件的公开成员函数

当我们需要调用组件的公开成员函数时，就需要使用组件的引用来调用。调用方式是：

```
this.refs.aReferName.公开成员函数名()
```

6.7.4 重新设定组件的属性

每一个 React Native 组件都有一个公开的成员函数 setNativeProps，使用它可以增加或者修改 React Native 组件的属性。

React Native 开发不建议使用 setNativeProps 函数。它是一个“简单、粗暴”的方法，可以直接操作任何层面组件的属性，而不是使用 React Native 组件的状态机变量，这样会使代码逻辑混乱，有可能打乱原来设计编写好的业务逻辑。在使用 setNativeProps 之前，请尽量先尝试用 setState 和 shouldComponentUpdate 方法来解决开发者的问题。

在不得不频繁刷新而又遇到性能瓶颈时，比如创建连续的动画，同时要避免渲染组件结构和同步太多的视图变化所带来的大量开销时，才考虑使用 setNativeProps 函数。

代码 6-14 示范了组件引用的使用。

代码 6-14：

```

'use strict';
let React = require('react-native');
let {
  AppRegistry, StyleSheet, View, Text, TextInput, Image
} = React;
let Project19 = React.createClass({
  getInitialState: function() {
    return {
      textInputValue: ''
    };
  },
  buttonPressed: function() { //当按钮被按下时执行此函数
    let textInputValue = 'new value';
    this.setState({textInputValue}); //改变 TextInput 组件中的字符串值
  }
});

```

```

        this.refs.textInputRefer.setNativeProps({ //修改文本输入框的属性值
            editable:false //使文本输入框变为不可编辑
        });
        this.refs.text2.setNativeProps({ //通过指向 Text 组件的引用
            style: { //修改该组件的颜色与字体大小
                color: 'blue',
                fontSize:30
            }
        });
    },
    render:function() {
        return (
            <View style={styles.container}>
                <Text style={styles.buttonStyle}
                    onPress={this.buttonPressed}>
                        Press me genterly
                    </Text>
                <Text style={styles.textPromptStyle}
                    ref='text2'> //指定本组件引用名
                        文字提示
                    </Text>
                <View>
                    <TextInput style={styles.textInputStyle}
                        ref='textInputRefer' //指定本组件引用名
                        value={this.state.textInputValue}
                        onChangeText={(textInputValue)=>this.setState({textInputValue})}/>
                </View>
            </View>
        );
    },
    });
const styles = StyleSheet.create({
    container: {
        flex: 1,
        justifyContent: 'center',
        alignItems: 'center',
        backgroundColor: 'white'
    },
    buttonStyle: { //文本组件样式,使用该文本组件作为简单的按钮
        fontSize:20,
        backgroundColor:'grey'
    },
    textPromptStyle: { //文本组件样式
        fontSize:20,
    },
    textInputStyle: { //文本输入框组件样式
        width:150,
        height:50,
        fontSize:20,
        backgroundColor:'grey'
    }
});
AppRegistry.registerComponent('Project19', () => Project19);

```

执行代码 6-14，当用户按下按钮后，会通过 `setNativeProps` 函数修改两个组件的属性，让它们产生视觉上的变化，这些变化可以通过并且应当通过状态机变量来完成。代码 6-14 完全是为了示范如何使用组件的引用，以及如何调用 `setNativeProps` 函数修改组件属性才不使用状态机变量，而使用 `setNativeProps` 函数来完成相应功能的。

6.7.5 获得组件的位置

每一个 React Native 组件都有一个 `measure` 成员函数，调用它可以得到组件当前的宽、高与位置信息。

获得组件宽、高及位置信息的例程如代码 6-15 所示。

代码 6-15:

```
'use strict';
var React = require('react-native');
var {
  AppRegistry, StyleSheet, TextInput, View,
} = React;
var Project19 = React.createClass({
  componentDidMount:function() {
    var aref = this.tempfunc;
    window.setTimeout( aref, 1 );    //在 componentDidMount 执行完后才可以获取位置
  },                                //因此指定一个 1 毫秒后超时的定时器
  tempfunc:function() {
    this.refs.aTextInputRef.measure( this.getTextInputPositon);    //获取位置
  },
  //位置信息通过回调函数传递给开发者
  getTextInputPositon:function(fx, fy, width, height, px, py) {
    console.log('getTextInputPositon');
    console.log('Component width is: ' + width);
    console.log('Component height is: ' + height);
    console.log('X offset to frame: ' + fx);    //这个值无用
    console.log('Y offset to frame: ' + fy);    //这个值无用
    console.log('X offset to page: ' + px);
    console.log('Y offset to page: ' + py);
  },
  render: function() {
    return (
      <View style={styles.container}>
        <View style={{borderWidth:1}}>
          <TextInput style={styles.textInputStyle}
            ref='aTextInputRef'
            defaultValue='Ajfg 你好'
            underlineColorAndroid='white' />
        </View>
      </View>
    );
  }
});
var styles = StyleSheet.create({
  container: {
```

```

        flex:1,
        justifyContent: 'center',
        alignItems: 'center',
        backgroundColor:'white'
    },
    textInputStyle: {
        width:200,
        height:55,
        fontSize:50,
        alignItems:'center',
        justifyContent: 'center',
        paddingTop:0,
        paddingBottom: 0
    }
  });
  AppRegistry.registerComponent('Project19', () => Project19);

```

使用 View 组件的 onLayout 回调函数是获取组件的宽、高与位置信息的好办法。但对某些代码生成的组件，使用组件的 measure 成员函数获取组件的宽、高及位置信息是唯一的方法。

6.8 跨平台状态栏组件

StatusBar 是 React Native 0.20.0 新增的跨平台组件，它可以用来设置并动态改变设备的状态栏显示特性。

6.8.1 StatusBar 组件属性

animated 是布尔类型的属性，用来设定状态栏的背景色、样式和隐现被改变时，是否需要动画效果。它的默认值是 false。

hidden 是布尔类型的属性，用来设定状态栏是否隐藏。

6.8.1.1 Android 平台特有属性

backgroundColor 是颜色类型的属性，用来设定 Android 设备上状态栏的背景颜色。

translucent 是布尔类型的属性，用来设定 Android 设备上的状态栏显示是否有半透明效果。如果设置为 true，React Native 应用将会从物理屏幕的顶端开始显示，而状态栏会浮现在应用顶部显示区域的上方。

6.8.1.2 iOS 平台特有属性

barStyle 是字符串类型的属性，它的取值为 default 和 light-content 之一，用来设定 iOS 设备状态栏的文字图标颜色。它的默认值是 default。

默认风格的 iOS 设备状态栏效果如图 6-19 所示。

明亮（light-content）风格的 iOS 设备状态栏效果如图 6-20 所示。开发者需要注意，如果状态

栏设置为明亮风格，那么它的背景色不能为白色；否则状态栏会与背景色融为一体，无法看到。

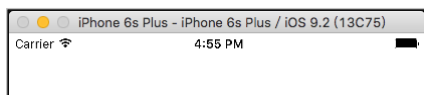


图 6-19 iPhone 手机默认风格状态栏



图 6-20 iPhone 手机明亮风格状态栏

`networkActivityIndicatorVisible` 是布尔类型的属性，用来设定网络活动指示器是否在状态栏显示。它的默认值为 `false`。如图 6-21 所示是 iPhone 手机打开网络活动指示器后的效果截图。注意“中国联通”四个字的右侧是扇形的 WiFi 已连接标志，其旁边就是网络活动指示器。当手机有网络数据交互时，它就会不停地旋转。

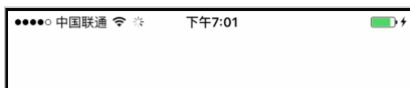


图 6-21 iPhone 手机网络活动指示器被打开后的效果截图

`showHideTransition` 是字符串类型的属性，它只可在 `fade` 和 `slide` 中取一个值。它决定使用 `hidden` 属性来显示与隐藏状态栏时的效果是淡入淡出还是滑入滑出。它的默认值是 `fade`。

6.8.2 StatusBar 组件使用示例

代码 6-16 简单地示范了 `StatusBar` 组件的使用。`StatusBar` 组件的使用比较简单，就是对几个属性设定值以决定状态栏的效果。

代码 6-16:

```
'use strict';
var React = require('react-native');
var {
  AppRegistry, StyleSheet, StatusBar, View
} = React;
var Project20 = React.createClass({
  render: function() {
    return (
      <View style={{flex: 1, backgroundColor: 'white',borderWidth:1}}>
        <StatusBar
          animated={true}
          hidden={false}
          backgroundColor='grey'
          translucent={true}
          barStyle='default'
          showHideTransition='fade'
          networkActivityIndicatorVisible={ true} />
      </View>
    );
  }
});
AppRegistry.registerComponent('Project20', () => Project20);
```

读者可以运行代码 6-16，并修改各属性的值，查看状态栏的改变情况。

如果应用需要，开发者可以将状态机变量指定为 `StatusBar` 的属性，然后在代码中通过设置状态机变量来控制状态栏的显示特性。

6.8.3 手机状态栏在开发中的处理

在 React Native 0.20.0 版本之前，因为没有提供 `StatusBar` 组件，开发者需要在 React Native 应用程序平台自适应时，考虑对不同平台的手机状态栏的处理。本节与 6.8.4 节描述的是在 0.20.0 版本前对手机状态栏处理的技术。开发者在阅读老代码时，可能会看到相关技术。

6.8.3.1 Android 手机状态栏

在 Android 平台上，当状态栏呈现在 Android 手机屏幕顶部时，它会占用“状态栏高度”× 当前屏幕宽度的空间，开发者只能使用剩下的屏幕空间。

如果要动态计算屏幕中组件的位置时，在 Android 平台上是手机屏幕的高度减去状态栏的高度，得到的才是开发者可以使用的屏幕高度。开发者从第 0 行开始放置组件时，组件会紧贴着状态栏的下边沿显示。

在 React Native 0.19.0 版本之前，是有办法可以得到 Android 手机状态栏的高度的。但自从 0.20.0 版本引入 `StatusBar` 插件后，原来的办法不能再使用了，并且通常也不需要去得到手机状态栏的高度了。开发者暂时无法通过 React Native 提供的能力得知 Android 设备的状态栏高度。但因为 Android 设备常用的屏幕分辨率规格不多，可以通过取得设备屏幕分辨率来得出 Android 设备的状态栏高度。比如对于最常见的 1080 × 1920 分辨率的设备，状态栏的高度是 25pt。

6.8.3.2 iPhone 手机状态栏

如果不通过 `StatusBar` 组件控制，在默认情况下，状态栏会显示在手机屏幕的最上方。

开发者在 iOS 平台取到的屏幕高度就是使用的高度。开发者从第 0 行开始排列组件时，组件会紧贴着手机屏幕的最上沿显示。此时如果状态栏没有被设置为隐藏，则会浮现在开发者排列在第 0 行的组件上方，如果在这部分屏幕空间开发者设计有可显示组件，则该组件会被部分遮盖。

如果不想设置状态栏为隐藏状态，则应当空出状态栏的显示区域，但可以为这个区域设置背景色，以使整个界面风格统一。

在 iOS 平台的 `StatusBar` 组件中，React Native 为开发者提供了 `getHeight` 静态函数以取得当前设备状态栏的高度。它的使用方法为：

```
.....
callback:function(aResult) {
  console.log('Height:'+aResult.height);
},
.....
StatusBar.getHeight( this.callback);
.....
```

6.8.4 StatusBarIOS API

在 React Native 0.20.0 版本之前，React Native 只为开发者提供了 StatusBarIOS API，供用户设置 iOS 手机的状态栏。对于 Android 设备的状态栏，则只能通过混合开发在原生代码中进行处理。

在 React Native 0.20.0 版本之后，开发者应当不再使用此 API。因为使用 StatusBar 组件，可以做到两个平台的代码统一。

在此列出此 API，只是考虑到读者有可能在阅读网上的旧代码时会遇到此 API。

StatusBarIOS API 提供了 setStyle 静态函数供开发者设置状态栏的风格。它的原型是：

```
static setStyle(style: StatusBarStyle, animated?: boolean)
```

通常开发者使用如下方式调用它：

```
.....
var {
    AppRegistry, StatusBarIOS, Navigator,    BackAndroid
} = React;
.....
if (StatusBarIOS !== undefined) { //如果在 Android 平台上运行，StatusBarIOS 等 undefined
    StatusBarIOS.setStyle( 0 );           //默认风格，状态栏图标为黑色
    StatusBarIOS.setStyle( 'default' );   //这条语句等价于上一条语句
}
.....
```

默认风格的手机状态栏显示效果如图 6-20 所示。

设置状态栏为明亮风格的代码如下：

```
.....
StatusBarIOS.setStyle( 1 );           //明亮风格，状态栏图标为白色
StatusBarIOS.setStyle( 'light-content' ); //这条语句等价于上一条语句
.....
```

开发者需要注意，如果状态栏设置为明亮风格，那么它的背景色不能为白色；否则状态栏会与背景色融为一体，无法看到。明亮风格的手机状态栏显示效果如图 6-21 所示。

setStyle 还可以有第二个布尔值参数，用来定义状态栏风格转换时是否有动画效果。有兴趣的读者可以自行在手机上运行代码，查看设置这个参数的效果。

StatusBarIOS API 提供了 setHidden 静态函数供开发者设置状态栏是否隐藏。它的原型是：

```
static setHidden(hidden: boolean, animation?: StatusBarAnimation)
```

通常开发者通过下面代码将状态栏设置为隐藏：

```
.....
StatusBarIOS.setHidden( true );
.....
```

StatusBarIOS API 提供了 setNetworkActivityIndicatorVisible 静态函数供开发者单独设置状态栏中的网络活动指示器。它的原型是：

```
static setNetworkActivityIndicatorVisible(visible: boolean)
```


开发者通过下面的代码打开网络活动指示器：

```
.....
StatusBarIOS.setNetworkActivityIndicatorVisible ( true );
.....
```

真机状态栏效果如图 6-21 所示。

6.9 高度自增长的扩展 TextInput 组件

在移动应用开发中，希望输入区域的高度随着输入内容的长度而增长是一个常见的功能需求。在 React Native 0.20.0 后，开发者已经可以实现这种组件了。

实现高度自增长的扩展 TextInput 组件代码参见代码 6-17。

代码 6-17，index.ios.js 或者 index.android.js：

```
'use strict';
var React = require('react-native');
var {
  AppRegistry, StyleSheet, Text, TextInput, View
} = React;

var AutoExpandingTextInput = React.createClass({ //自定义高度动态调整的组件
  getInitialState:function() {
    return {text: '', height: 0};
  },
  _onChange:function(event) {
    this.setState({
      text: event.nativeEvent.text,
      height: event.nativeEvent.contentSize.height,
    });
  },
  render: function() {
    return (
      <TextInput {...this.props} //将自定义组件的所有属性交给 TextInput
        multiline={true}
        onChange={this._onChange}
        style={[styles.textInputStyle, {height: Math.max(35, this.state.height)}]}
        value={this.state.text}/>
    );
  }
});

var Project20 = React.createClass({
  _onChangeText:function(newText) {
    console.log('inputed text:' + newText);
  },
  render: function() {
    return (
      <View style={styles.container}>
        <AutoExpandingTextInput style={styles.textInputStyle}
          onChangeText={this._onChangeText}/>
      </View>
    );
  }
});
```

```

    }
  });
  var styles = StyleSheet.create({
    container: {
      flex: 1,
      justifyContent: 'center',
      alignItems: 'center',
      backgroundColor: '#F5FCFF',
    },
    textInputStyle: {
      fontSize: 20,
      width: 300,
      height: 30,
      backgroundColor: 'grey',
      padding: 0,
    }
  });
  AppRegistry.registerComponent('Project20', () => Project20);

```

代码 6-17 运行效果参见图 6-22。



图 6-22 代码 6-17 运行效果

如图 6-22 所示，当用户的输入超过 `TextInput` 组件的高度时，`TextInput` 组件将会自动变高，把用户的所有输入都显示出来。

6.10 访问操作系统剪贴板

React Native 为开发者提供了 Clipboard API，让开发者可以访问设备操作系统中剪贴板中的内容，或者向剪贴板中存放内容。目前还只支持获取或者存放字符串。

Clipboard API 比较简单，只有两个静态函数：`setString` 向剪贴板中存放字符串；`getString` 从剪贴板中取出字符串。它的使用请参见代码 6-18。

代码 6-18, index.android.js 或者 index.ios.js:

```
'use strict';
var React = require('react-native');
var {
  AppRegistry, StyleSheet, Text, View, Clipboard //导入 Clipboard API
} = React;
var Project20 = React.createClass({
  getInitialState: function() {
    return {
      textFromClipboard: ' '
    };
  },
  pasteFromClipboard: function() {
    Clipboard.getString().then( (textFromClipboard)=> { //从剪贴板中读取字符串
      this.setState({textFromClipboard});
    }).catch((error)=>{
      console.log('error:'+error);
    });
  },
  copyToClipboard: function() {
    Clipboard.setString('ABCD 你好'); //向剪贴板中存入字符串
  },
  render: function() {
    return (
      <View style={styles.container}>
        <Text style={styles.welcome}>
          {this.state.textFromClipboard}
        </Text>
        <Text style={styles.instructions}
          onPress={this.copyToClipboard}>
          Press to Copy something to Clipboard.
        </Text>
        <Text style={styles.instructions}
          onPress={this.pasteFromClipboard}>
          Press to Paste.
        </Text>
      </View>
    );
  }
});
var styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#F5FCFF',
    borderWidth: 1
  },
  welcome: {
    fontSize: 20,
    textAlign: 'center',
    margin: 10,
  },
  instructions: {
```

```
      textAlign: 'center',  
      color: '#333333',  
      marginBottom: 15,  
      backgroundColor: 'grey',  
      fontSize: 30  
    },  
  });  
AppRegistry.registerComponent('Project20', () => Project20);
```

运行代码后，比较方便的测试手段是进入手机的短信程序，长按某条短信，然后选择“复制文本”菜单项，这样这条短信就被复制到剪贴板中。然后进入 Project20 应用，点击“Press to Paste”按钮，就可以在屏幕上看到被粘贴的文字，文字内容正是刚才复制的短信内容。运行效果参见图 6-23。



图 6-23 代码 6-18 运行效果

点击“Copy something to Clipboard”按钮，就可以把“ABCD 你好”这个字符串放入剪贴板中，然后再进入其他应用的编辑界面（比如编辑短信），长按输入框，选择粘贴，就可以看到“ABCD 你好”被粘贴到其他应用的输入框中。

第 7 章

组件生命周期、数据存储及 React Native 应用实现步骤

7.1 组件生命周期

一个 React Native 组件从它被 React Native 框架加载，到最终被 React Native 框架卸载，会经历一个完整的生命周期。在这个生命周期中，开发者可以定义一些生命周期函数，用来处理在特定条件下 React Native 组件将要执行的操作，比如在某个时间点读取数据等。

在前面的章节中，我们已经接触到了一些生命周期函数，在这里将对它们进行详细、完整的讨论。

7.1.1 `getInitialState`

其函数原型是：

```
object getInitialState()
```

这个函数将在 React Native 组件被加载前调用一次。它的返回值会成为 `this.state` 的初始值。

7.1.2 `getDefaultProps`

其函数原型是：

```
object getDefaultProps()
```

这个函数在组件被创建时调用一次。它的返回值会成为 `this.props` 的初始值。在这之后，如果父组件指定了组件 `props` 中的某些值，这些值将会与 `this.props` 的初始值合并，如果有相同的键，父组件指定的键将覆盖初始值中的键。

细节：`getDefaultProps()` 返回的任何复杂对象都将会在组件的各个实例间共享，而不是每个实例拥有一份拷贝。通过这种方法，做到资源的节约。

7.1.3 componentWillMount

其函数原型是：

```
componentWillMount()
```

在 React Native 组件的生命周期中，这个函数只会被执行一次。它在初始渲染（render 函数被 React Native 框架调用执行）前被执行，当它执行完后，render 函数会马上被 React Native 框架调用执行。如果在这个函数里调用 setState 函数改变了某些状态机变量的值，React Native 框架不会执行渲染操作，而是等待这个函数执行完成后再执行初始渲染。

React Native 组件的子组件也有 componentWillMount 函数，并且会在父组件的 componentDidMount 函数之后被调用。

这个函数无参数并且不需要任何返回值。

如果开发者需要从本地存储中读取数据用于显示，那么在这个函数里进行读取是一个很好的时机。

7.1.4 componentDidMount

其函数原型是：

```
componentDidMount()
```

在 React Native 组件的生命周期中，这个函数只会被执行一次。它在初始渲染执行完成后会马上被调用。在 React Native 组件生命周期的这个时间点之后，开发者可以通过子组件的引用来访问、操作任何子组件。React Native 组件的子组件也有 componentDidMount 函数，并且会在父组件的 componentDidMount 函数之前被调用。

这个函数无参数并且不需要任何返回值。

如果 React Native 应用需要在程序启动显示初始界面后从网络侧获取数据，那么把从网络侧获取数据的代码放在这个函数里是一个不错的选择。

7.1.5 componentWillReceiveProps

这个函数的原型是：

```
componentWillReceiveProps(object nextProps)
```

React Native 组件的初始渲染执行完成后，当 React Native 组件接收到新的 props 时，这个函数将被调用。这个函数接收一个 object 参数，object 里是新的 props。

注意：当 React Native 初次被渲染时 componentWillReceiveProps 函数并不会被触发。这种机制是故意设计的。

如果新的 props 会导致界面重新渲染，这个函数将在渲染前被执行。在这个函数中，老的 props 可以通过 this.props 访问，新的 props 在传入的 object 中。如果在这个函数中通过调用 this.setState

函数改变某些状态机变量的值，React Native 框架不会执行对这些状态机变量改变的渲染，而是等 `componentWillReceiveProps` 函数执行完成后一起渲染。

这个函数不需要返回值。

7.1.6 `shouldComponentUpdate`

这个函数的原型是：

```
boolean shouldComponentUpdate(object nextProps, object nextState)
```

React Native 组件的初始渲染执行完成后，当 React Native 组件接收到新的 `state` 或者 `props` 时，这个函数将被调用。它接收两个 `object` 参数，其中第一个是新的 `props`；第二个是新的 `state`。这个函数需要返回一个布尔值，告诉 React Native 框架针对这次改变，React Native 是否需要重新渲染本组件。如果此函数返回 `false`，React Native 将不会重新渲染本组件，相应的，本组件的 `componentWillUpdate` 和 `componentDidUpdate` 函数也不会被调用。

React Native 组件默认的 `shouldComponentUpdate` 函数总是返回 `true` 值。如果开发者遵从了 2.5.2 节提到的视状态机变量为“不可变的常量”这个 React Native 开发规则，那么开发者可以提供自己的 `shouldComponentUpdate` 函数，在函数中比较新老版本的 `state` 和 `props`，判断是否需要重新渲染。

通过这个函数来阻止无必要的重新渲染，是提高 React Native 应用程序性能的一大技巧。

7.1.7 `componentWillUpdate`

这个函数的原型是：

```
componentWillUpdate(object nextProps, object nextState)
```

React Native 组件的初始渲染执行完成后，React Native 框架在重新渲染 React Native 组件前会调用这个函数。开发者可以在这个函数中为即将发生的重新渲染做一些准备工作，但开发者不能在这个函数中通过 `this.setState` 再次改变状态机变量的值。如果需要改变，则在 `componentWillReceiveProps` 函数中进行改变。

这个函数不需要返回值。

7.1.8 `componentDidUpdate`

其函数原型是：

```
componentDidUpdate(object prevProps, object prevState)
```

React Native 组件的初始渲染执行完成后，React Native 框架在重新渲染 React Native 组件完成后会调用这个函数。传入的两个参数是渲染前的 `props` 和 `state`。

这个函数不需要返回值。

7.1.9 componentWillUnmount

其函数原型是：

```
componentWillUnmount()
```

在 React Native 组件被卸载前，这个函数将被执行。这个函数没有参数，也不需要返回值。

如前面章节所述，如果 React Native 组件申请了某些资源或者订阅了某些消息，那么需要在这个函数中释放资源，取消订阅。

7.2 读取 JSON 文件

在编写代码时，开发者有时可能需要存储一些比较大的、在应用程序运行时不需要更改的数据。因为大，不便于直接写在代码中，笔者推荐使用 JSON 文件存储这些数据。

这种做法有两个优点：

- 数据存放在单独的文件中，使得代码精简、有条理。
- JSON 数据格式便于阅读、修改。

在项目目录下建立一个名为 data 的目录，然后在 data 目录下创建一个名为 SimpleSample.json 的文本文件。它的内容见代码 7-1。

代码 7-1：

```
{
  "employees": [{
    "givenName": "三",
    "FamilyName": "张",
    "salary": 1,
  }, {
    "givenName": "四",
    "FamilyName": "李",
    "salary": 2,
  }, {
    "givenName": "二",
    "FamilyName": "王",
    "salary": 3,
  }, ]
}
```

在这个 JSON 文件中，在每个 JSON 对象中都刻意使用了尾逗号（trailing commas）。使用 React Native 的 require 语句读取 JSON 数据时，支持有尾逗号的 JSON 数据。

请读者注意 JSON 文件的文件名后缀必须是小写的。

当我们需要使用 JSON 文件中的数据时，在代码中加入：

```
let constantData = require('./data/SimpleExample.json');
```

现在就可以将 constantData 变量作为一个类的对象来使用了。示例如代码 7-2 所示。

代码 7-2:

```
console.log("constantData's type:" + typeof(constantData));
console.log("employees'type:" + typeof(constantData.employees));
console.log("employees' length:" + constantData.employees.length);
console.log("No.1's givenName:" + constantData.employees[0].givenName );
console.log("No.1's salary:" + constantData.employees[0].salary);
console.log("Type of No.1's salary:" + typeof(constantData.employees[0].salary));
```

以上语句在 React Native Dev tool 上的输出结果（每行前面的路径名视项目目录不同而不同）如下：

```
Page3.js:16 constantData's type:object
Page3.js:17 employees'type:object
Page3.js:18 employees' length:3
Page3.js:19 No.1's givenName:三
Page3.js:20 No.1's salary:1
Page3.js:21 Type of No.1's salary:number
```

7.3 数据持久化操作

数据持久化是指应用程序将某些数据存储在手机存储空间中。React Native 框架不支持调用 JavaScript 的 fs 包进行文件读写操作。React Native 框架为开发者提供了 AsyncStorage API，开发者可以利用它将任意“字符串键值对”保存到存储空间中，也可以通过键值将指定的字符串值从存储空间中取出。

AsyncStorage 不提供索引、排序等数据库中经常使用的功能。它只是一个简单的、异步的“键-值”存储系统，开发者可以用它来取代 Android 的 sharedperference 和 iOS 的 NSUserDefaults。如果需要使用数据库功能，开发者只能通过混合开发，利用不同平台的原生代码来获得数据库功能。

通过 AsyncStorage 的静态方法，数据会保存到手机存储空间中。开发者不需要指定，也不需要关心数据保存在了什么文件或者数据库、表中，AsyncStorage 存储的数据对该 React Native 应用都是全局可访问的，开发者只需要知道能通过 AsyncStorage 增、删、改、查数据就可以了。

每一个 AsyncStorage API 提供的方法都会返回一个 JavaScript 的 Promise 对象。如果读者对 JavaScript 的 Promise 机制不熟悉，则可以参考附录 A.5。

7.3.1 flow 语法检查器

在 React Native 官方文档上，AsyncStorage 类的所有静态函数都是以 flow 语法描述的。flow 是由 Facebook 开发的，是开源的静态类型检查器。以 multiGet 函数声明为例：

```
static multiGet(keys: Array<string>, callback?: ?(errors: ?Array<Error>,
result: ?Array<Array<string>>) => void)
```

flow 语法声明：

- multiGet 的第一个参数 keys 是一个 string 类型的数组，不能为 null 或者 undefined。
- callback?声明 multiGet 函数的第二个参数可以没有；如果有，它是一个不需要返回值的

回调函数。

- `errors: ?Array<Error>` 声明回调函数的第一个参数可以为 `null` 或者 `undefined`，第一个参数 `errors` 的类型是 `Error` 类的数组。
- `result: ?Array<Array<string>>` 声明回调函数的第二个参数类型是 `Array<Array<string>>`。它有可能为 `null` 或者 `undefined`。
- `?(errors: ?Array<Error>, result: ?Array<Array<string>>)` 声明回调函数的两个参数可以没有。

在官方文档中其他几个函数的描述含义都可以依此类推。在下面的讨论中，将不给出官方文档的 `flow` 语法描述。

7.3.2 写入数据、错误捕捉

开发者可以通过 `AsyncStorage` 类的静态函数 `setItem` 来存储数据。`setItem` 的 JavaScript 原型是：

```
static object setItem(key, value ,aCallback)
```

最简单的用法，提供回调函数的用法与通过 `Promise` 机制后续处理的方法示例见代码 7-3。

代码 7-3：

```
.....
AsyncStorage.setItem( 'name', '张三');    //最简单的用法，无法检测保存何时结束、是否成功
.....
doSomething: function(error) {              //定义保存结束后的处理函数
    if ( error != null ) {                  //有错误发生，处理错误
        console.log('error message:' + 'error.message');
        return;
    }
    //执行保存成功后的操作
},
.....
AsyncStorage.setItem('name', '张三',this.doSomething); //提供保存后回调的用法
.....
AsyncStorage.setItem('name', '张三').then(    //使用 Promise 机制的方法
    ()=>{                                     //定义操作成功的处理函数
        //这里写当数据保存成功后需要做的操作
    },
    (error)=>{                                //定义操作失败的处理函数，可以不提供
        console.log(' error:' + error.message);
    }
    );
.....
```

使用 `AsyncStorage` 类的静态函数时，开发者一定要注意提供的参数类型要正确（提供的键和值必须是字符型）。如果参数类型提供错误，JavaScript 的 `try...catch` 机制将无法捕捉到这个错误。

当提供的参数类型错误时，使用回调函数，在 `Promise` 机制的 `rejection` 状态处理函数中或者利用 `Promise` 机制发生错误时的回调函数都可以捕捉到错误。但这三种机制中的任何一种都不能阻止手机屏幕上弹出大红屏错误提示，同时程序运行中断。

上两段的描述对所有的 `AsyncStorage` 类的静态函数都有效。

代码 7-4:

```

.....
try {
    AsyncStorage.setItem('name',123);
} catch(error) {
    //JavaScript 的 try_catch 机制无法捕捉到错误
    console.log(' catch a error: '+error);
}
.....
.....
AsyncStorage.setItem('name',123).then(
    //使用 Promise 机制的方法
    ()=>{
        //定义操作成功的处理函数
        .....
        //当数据保存成功后需要做的操作
    }
    //不定义 rejection 状态处理函数,使用 Promise 机制发生错误时的回调函数
).catch((error)=>{
    //定义操作失败或者操作成功后进行抛出异常的处理
    console.log(' error:' + error.message);
    .....
    //对错误进行处理
});
.....

```

运行代码 7-4, 第一段 catch 不会抓到任何抛出的错误, 手机屏幕上会弹出大红屏错误信息。

代码 7-4 的第二段演示了如何定义 Promise 机制发生错误时的回调函数。这种方法仍然不能阻止手机屏幕弹出红色错误提示, 程序运行中断。

代码 7-4 的第二段演示的 catch 机制不仅可以捕捉到 AsyncStorage 类的静态函数抛出的异常, 还可以捕捉到操作成功时处理函数抛出的异常。开发者在实际开发过程中, 如果不能保证自己的处理函数绝对正确, 则应当使用这种 catch 机制。在使用这种机制时, 有可能需要判断抛出的异常是对象, 还是某个类型对象的数组 (AsyncStorage 类的静态函数抛出的异常), 然后分别进行处理。

开发者可以通过 AsyncStorage 类的静态函数 multiSet 来一次存储多个数据。它的 JavaScript 原型是:

```
static object multiSet(aArray ,aCallback)
```

最简单的用法, 提供回调函数的用法与通过 Promise 机制后续处理的方法示例见代码 7-5。注意, 如果 multiSet 抛出异常, 则抛出的异常是一个数组。

代码 7-5:

```

.....
//最简单的用法, 无法知道保存何时结束、是否成功
AsyncStorage.multiSet([['1', '张三'], ['2', '李四']]);
.....
doSomething: function(errors) {
    //提供保存后回调的用法, 先定义回调处理函数
    if (errors!= null ) {
        //有错误发生, 处理错误
        console.log(' error's length:' + errors.length);
        console.log('1st error message: ' + errors [0].message);
        .....
        return;
    }
    .....//执行保存成功后的操作
},

```

```

.....
AsyncStorage.setItem([[ '1', '张三'], [ '2', '李四']],this.doSomething);
.....
//使用 Promise 机制的方法
AsyncStorage.setItem([[ '1', '张三'], [ '2', '李四']]).then(
  ()=>{
    .....
    //定义操作成功的处理函数，操作成功时调用的函数没有参数
    //这里写当数据保存成功后需要做的操作
  }).catch( (errors)=>{
    .....
    //操作失败或者对异常的处理
    console.log(' error's length:' + errors.length);
    if ( errors.length > 0 ) {
      .....
      //保存操作有异常
      console.log('1st error message: ' + errors [0].message;
      .....
    }
    else {
      .....
      //异常不是数组，有可能是成功操作的处理函数抛出的异常
      //处理异常
    }
  });
.....

```

使用这两种方法之一保存数据时，如果保存的键已经存在本地存储中，将会用新的值覆盖原来的键对应的值。

7.3.3 读取数据

开发者可以通过 AsyncStorage 类的静态函数 getItem 来存储数据。它的 JavaScript 原型是：

```
static object getItem(aKey,aCallback)
```

开发者读取数据时，必须要在提供回调函数的用法和通过 Promise 机制后续处理的方法中选择。示例见代码 7-6。

代码 7-6:

```

.....
handleResult: function(error,result) {
  if( error != null) {
    .....
    //读取操作失败
    console.log('error message:' + error.message);
    .....
    //处理读取操作失败的代码
  }
  return;
}
If ( result===null ) {
  .....
  //存储中没有指定键对应的值，处理这种情况
  return;
}
console.log('result: ' + result);
.....
//正确获得了指定键对应的值，这里写处理 result 的代码
},
.....
AsyncStorage.getItem('name', this. handleResult);
//提供读取后回调函数的用法
.....
AsyncStorage.getItem('name').then( //使用 Promise 机制的方法
  (result)=>{
    .....
    //注意使用 Promise 机制时，如果读取操作成功，则不会有 error 参数
    If (result===null ) {
      .....
      //存储中没有指定键对应的值，处理这种情况
    }
    return;
  }
)

```

```

    }
    ..... //正确获得了指定键对应的值，这里写处理 result 的代码
  }).catch( (error)=>{ //读取操作失败，或者正确读取后的操作有异常抛出
    console.log(' error:' + error.message);
    ..... //处理读取操作失败的代码
  });
  .....

```

开发者可以通过 AsyncStorage 类的静态函数 `getAllKeys` 来得到当前存储的所有键。其函数原型是：

```
static object getAllKeys([aCallback])
```

开发者读取数据时，必须要在提供回调函数的用法与通过 Promise 机制后续处理的方法中选择。使用回调函数的示例见代码 7-7。注意代码 7-7 中两种机制得到的 `keys` 是一个字符串数组，数组中的每一个字符串都是一个键。然后开发者就可以通过 `getItem` 函数获取每一个键对应的值了。

代码 7-7:

```

.....
handleAllkeys: function(error,keys) {
  if (error != null ) { //获取所有的键值失败
    console.log('dataLoaded error: ' + error.message);
    return;
  }
  else console.log('get all key error is null. ');
  let arrayLen = keys.length; //回调函数得到的 keys 是 string 数组，处理这个数组
  for(let counter=0; counter < arrayLen; counter++ ) {
    console.log( 'key ' + counter + ':' + keys[counter]);
    AsyncStorage.getItem( keys[counter] ).then( //读取每一个 key 对应的值
      (result)=>{
        console.log('key '+keys[counter]+ 'getItem data:' + result );
      }).catch( (error)=> {
        ..... //对出错和异常进行处理
      });
  }
},
.....
AsyncStorage.getAllKeys(this.handleAllkeys); //提供读取后回调函数的用法
.....

```

运行代码 7-7，输出如下：

```

1 index.ios.js:304 get all key error is null.
2 index.ios.js:307 key 0:3
3 index.ios.js:307 key 1:1
4 index.ios.js:307 key 2:2
5 index.ios.js:307 key 3:7
6 index.ios.js:311 key 3 getItem data:333
7 index.ios.js:311 key 1 getItem data:111
8 index.ios.js:311 key 2 getItem data:222
9 index.ios.js:311 key 7 getItem data:777888

```

请读者注意，for 循环中的打印语句应当是先打印一个获取到的键值，然后打印该键对应的值，接下来再打印下一个键。但在输出中，第 2~5 行先把 4 个获取到的键值打印出来，然后第 6~8

行再打印每个键对应的值。为什么这样呢？因为 AsyncStorage 是异步获取的，当代码发出读取某键对应的值后，并没有暂停执行等待读取这个操作完成，而是继续向下执行，所以输出才是上面的结果。

从代码 7-7 的运行输出结果我们也可以清楚地观察到，使用 Promise 机制与使用回调函数一样，后续操作都是异步执行的。

代码 7-8 演示了如何使用 Promise 机制获取所有的键值。

代码 7-8:

```
.....
AsyncStorage.getAllKeys().then(                                //使用 Promise 机制的方法
  (keys)=>{                                                    //这里写处理 keys 的代码
    .....
  }).catch( (error)=>{
    .....                                                    //处理失败或者异常
  });
.....
```

开发者可以通过 AsyncStorage 类的静态函数 multiGet 得到指定的多个键对应的多个值。其函数原型是：

```
static object multiGet(aArray,aCallback)
```

假设本地存储已经通过代码 7-5 存储了['1', '张三'], ['2', '李四']这两对数据。代码 7-9 演示了如何通过 multiGet 将这两对数据取出。

代码 7-9:

```
.....
handleAllResults: function(errors,results) {
  if (errors!= null) {
    console.log(' error's length:' + errors.length);
    console.log('1st error message:'+ errors [0].message;
    .....                                                    //获取多个值失败，在这里写处理代码
    return;
  }
  console.log(results[0][0]);                                //打印出 1
  console.log(results[0][1]);                                //打印出张三
  console.log(results[1][0]);                                //打印出 2
  console.log(results[1][1]);                                //打印出李四
},
.....
AsyncStorage.multiGet(['1','2'],this.handleAllResults);      //提供读取后回调函数
.....
AsyncStorage. multiGet(['1','2']).then(                      //使用 Promise 机制
  (results)=>{                                                //获取成功
    console.log(results[0][0]);                                //打印出 1
    console.log(results[0][1]);                                //打印出张三
    console.log(results[1][0]);                                //打印出 2
    console.log(results[1][1]);                                //打印出李四
    .....
  }).catch( (errors)=>{                                       //获取失败
```

```

        console.log('error's length:' + errors.length);
        console.log('1st error message:' + errors [0].message;
        .....
        //获取多个值失败，在这里写处理代码
    });
    .....

```

开发者可以通过 AsyncStorage 类的静态函数 flushGetRequests 取消前面所有未执行完成的 multiGet 操作。其函数原型是：

```
static object flushGetRequests()
```

它可以通过 Promise 机制来进行取消后的处理。示例代码见代码 7-10。

代码 7-10:

```

.....
AsyncStorage.flushGetRequests().then(           //使用 Promise 机制
    ()=>{
        .....
        //取消操作成功后需要执行的操作
    }).catch((error)=>{
        console.log(' error:' + error.message);
        .....
        //取消操作失败后需要执行的代码
    });
.....

```

7.3.4 AsyncStorage API 存储数据的无序性

开发者很关心的一个问题就是使用 AsyncStorage API 存储数据，是有序的吗？答案很让人遗憾。AsyncStorage API 存储的数据是无序的。按一定顺序存储进去的数据，下一次读出时有可能乱序。开发者不能依赖 AsyncStorage API 保证存储数据的有序性，如果有需要，只能自行通过代码来实现排序。

7.3.5 删除数据

开发者可以通过 AsyncStorage 类的静态函数 removeItem 删除存储中某个指定的键及其对应的值。其函数原型是：

```
static object removeItem(aKey[,aCallback])
```

最简单的用法，提供回调函数的用法与通过 Promise 机制后续处理的方法示例见代码 7-11。代码 7-11 将代码 7-4 保存的数据删除掉。

代码 7-11:

```

.....
AsyncStorage.removeItem('name');           //最简单的用法，不知道删除什么时候完成
.....
doSomething: function(error) {             //删除成功后的回调函数
    if ( error != null ) {                 //有错误发生，处理错误
        console.log('error message:' + 'error.message');
        return;
    }
    .....
    //删除成功后需要执行的操作
}

```

```

},
.....
AsyncStorage.removeItem('name',this.doSomething);    //删除后执行回调函数的用法
.....
AsyncStorage.removeItem('name').then(                //使用 Promise 机制的方法
    ()=>{                                           //定义操作成功的处理函数
        //这里写当数据删除成功后需要做的操作
    }).catch((error)=>{                            //操作失败或者成功处理抛出异常
        console.log(' error:' + error.message);
        .....                                     //处理异常
    });
.....

```

开发者可以通过 AsyncStorage 类的静态函数 clear 删除数据存储中所有的键及其对应的值。其函数原型是：

```
static object clear([aCallback])
```

它可以通过 Promise 机制来进行取消后的处理。示例代码见代码 7-12。

代码 7-12:

```

.....
    AsyncStorage.clear();                                //最简单的用法，无法知道什么时候完成
.....
doSomething: function(error) {                          //全部删除成功后的回调函数
    if ( error != null ) {                               //有错误发生，处理错误
        console.log('error message:' + 'error.message');
        return;
    }
    .....                                              //全部删除成功后需要执行的操作
},
.....
AsyncStorage.clear(this.doSomething);                  //全部删除后执行回调函数的用法
.....
AsyncStorage.clear().then(                             //定义全部删除成功的处理函数
    ()=>{                                              //这里写当全部数据删除成功后需要做的操作
    }).catch( (error)=>{                               //操作失败或者成功处理抛出异常
        console.log(' error:' + error.message);
        .....                                       //处理异常
    });
.....

```

开发者可以通过 AsyncStorage 类的静态函数 multiRemove 删除数据存储中指定的多个键及其对应的值。其函数原型是

```
static object multiRemove(aArray[,aCallback])
```

假设本地存储已经通过代码 7-5 存储了['1', '张三'], ['2', '李四']这两对数据。代码 7-13 演示了如何通过 multiRemove 将这两对数据删除。

代码 7-13:

```

.....
doSomething: function(errors) {

```



```

if (errors!= null) {
    console.log(' error's length:' + errors.length);
    console.log('1st error message:'+ errors [0].message;
    .....
}
.....//删除成功后需要执行的操作
},
.....
AsyncStorage.multiRemove(['1', '2'],this.doSomething); //提供读取后回调函数
.....
AsyncStorage. multiRemove(['1', '2']).then(
    ()=>{
        //这里写当数据删除成功后需要做的操作
    }).catch( (errors)=>{
        //操作失败或者成功处理抛出异常
        console.log('error's length:' + errors.length);
        if( erroes.length > 0 ) {
            console.log('1st error message:'+ errors [0].message;
            .....
        }
        else {
            .....
        }
    });
.....

```

7.3.6 修改数据

前面已经提到，使用 AsyncStorage 类的静态方法写入数据时，后写入的相同键值的值会覆盖原有的值。建议开发者使用这种方法来修改数据。

AsyncStorage 类提供了 mergeItem 和 multiMerge 来实现修改数据。但官方文档声明这两个方法还不能够跨平台实现。这里只列出它们的函数原型：

```

static object mergeItem(aKey, aValue, aCallback)

static object multiMerge(aArray, aCallback)

```

7.3.7 JSON 对象存储

使用 AsyncStorage 类来保存数据，只能通过键-值形式保存一个字符串类型的数据。这很难满足开发者的需求。但配合 JSON 类，就可以满足开发者保存不同类型、多个变量数据的需求。

利用 JSON 类的 stringify 静态函数可以将 JSON 格式的对象（7.2 节中转化而来的对象就是 JSON 格式的对象）转化为一个字符串，然后通过 AsyncStorage 类的键-值形式进行保存。假设 constantData 是代码 7-1 描述的 JSON 文件通过 require 语句转化而来的对象（代码 7-2 演示了如何访问该对象），代码 7-14 演示了如何将这个对象再转化为字符串。

代码 7-14：

```

let constantData = require('./data/SimpleExample.JSON');
let newJSONString=JSON.stringify(constantData);

```

7.3.8 读取 JSON 对象

使用 7.3.7 节中讨论的方法保存的 JSON 对象，将它从存储中读取出来并转化为 JSON 对象也很容易。JSON 类的 `parse` 静态函数可以让开发者轻松地实现这个功能，示例见代码 7-15。

代码 7-15:

```
.....
let anotherData=JSON.parse(newJSONString);
.....
```

需要注意的是，在 7.2 节中，我们看到 React Native 使用 `require` 语句处理 JSON 格式文件时，JSON 文件中的字符串可以有尾逗号，但 JSON 类的 `parse` 函数要求严格，不允许有尾逗号。如果有尾逗号，`parse` 函数将抛出异常。比如下面的代码就会抛出异常：

```
JSON.parse('{ "name": "张三", }');
```

7.4 数据表操作

对于一些简单的设置之类的保存，7.3 节中讨论的功能已经能满足需求了，但对于复杂的数据表来说还不够。

当开发者需要进行类似于数据表的操作时，首先要明确使用数据表的操作，以及使用何种方式来实现。实现方式分为两种：通过 `AsyncStorage` API 实现和通过原生代码实现。

通过 `AsyncStorage` API 实现的思路是，在程序启动时数据表就全部被读入内存，之后的增、删、改、查、排序操作都是对内存中的数据表进行操作的，同时与手机存储保持同步。

我们将在 7.5 节的日记例程（上）中看到如何利用 `AsyncStorage` API 实现数据表操作。

7.5 React Native 应用实现步骤、日记例程（上）

在本节中，将讨论如何实现一个日记例程，用户可以通过这个应用来查看日记、搜索日记和新建日记。

因为这个日记例程需要使用到第 8 章才会介绍的 `ScrollView` 和 `ListView` 组件，所以该例程分为上、下两个部分。在上部分中，我们专注于 React Native 应用实现的常用步骤，会使用一些伪数据来实现一些功能。

我们将使用 2.5.3 节提及的 React Native 应用设计思路来实现这个例程。创建多个只负责渲染数据的无状态的 React Native 组件，将它们封装在一个有状态的 React Native 组件中，并把这个有状态的 React Native 组件的状态机变量的值通过 `props` 传给无状态的 React Native 组件（这时这些无状态的 React Native 组件是有状态的 React Native 组件的子组件）。在这种设计思路下，有状态的 React Native 组件封装了 UI 的交互逻辑，而无状态的 React Native 组件负责渲染 UI 界面。

在整个应用设计中，始终按照自下而上的原则进行。在大型的项目中，自下而上的设计方式简单，可以并行工作，并且可以在构建的同时写测试用例。

在实现过程中讨论了 React Native 设计的 5 个步骤，请大家一定要仔细阅读、思考、理解。把例程上部分实现的整个过程理解透了，React Native 开发就算是入门了。

7.5.1 应用原型

图 7-1 给出的是应用主界面设计图草图（正规的设计图需要提供各种距离，以及各部分的不同色值）。

- 当用户输入搜索关键字时，在日记列表中自动列出标题含有关键字的日记；
- 点击“写日记”按钮后，进入写日记界面；
- 日记列表按时间倒序排列。每个列表项左侧的图片用来表示心情。在设计图中，使用了一些简单的图片；
- 点击日记列表中的某一项，进入查看该日记界面。

阅读日记界面如图 7-2 所示。点击“返回”按钮将返回至主界面；点击“上一篇”按钮将在本界面中显示上一篇日记；点击“下一篇”按钮将显示下一篇日记。



图 7-1 日记列表界面



图 7-2 阅读日记界面

写日记界面如图 7-3 所示。点击“返回”按钮将不保存所编写的日记，直接返回主界面。点击“保存”按钮，将用户输入的标题、选择的心情、输入的日记正文，以及当前的时间保存在手机存储中，然后返回主界面。

界面上部的选择心情按钮被点击后，因为有 5 个心情选项，无法使用 Alert API 来实现，而应当使用 3.5.1 节中讨论的内容来实现多选界面。在这里改为用户每点击一次按钮，按钮上的心情提示语直接变一次，通过多次点击这个按钮，在 5 种心情中选择。

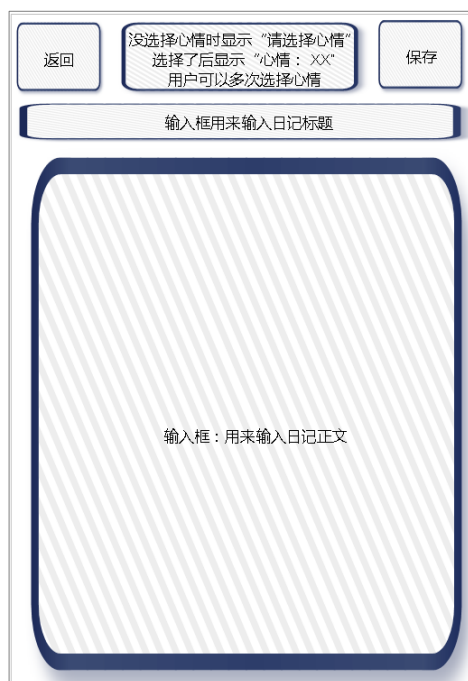


图 7-3 写日记界面

7.5.2 基础组件结构设计

至此，每个界面的原型设计已经完成了。现在需要设计每个界面的 React Native 基础组件的分层结构。

7.5.2.1 列出所有需要使用的基础组件

首先要做的是从 UI 界面上找出所有需要使用的基础组件。在这一步，我们能很容易地从界面上发现需要显示的图片、按钮、输入框、列表等，其中包括自定义组件。

基础组件需要遵循“责任唯一原则”。意思是一个基础组件只应当负责一个工作，如果一个组件负责多个工作，就应当将这个组件拆分成多个子组件。

7.5.2.2 对基础组件进行分层

对基础组件进行分层，目的是为了使用功能强大的 flexbox 布局来实现界面。分层就是把显示在同一个行区域或者列区域中的多个组件放在同一个 View 中。

比如在这个例程中，三个界面的最上方都是按钮（或者输入框加按钮），它们有相同的高度，就应当把它们放在同一个 View 中，分在同一个显示层次里。

开发者需要在界面上画出一个个方框，每个方框对应一个 flexbox 样式的 React Native 组件。需要在方框中再画出小方框，对应组件的子组件，直到无法画出更小的方框。这时，就得到了界

面对应的组件分层结构。

从主界面的原型设计中可以转化出如下的组件结构：

```
FilterableDiaryList (根 View)
  FirstRow(View)
    KeywordInputBox (TextInput)
    WriteDiaryButton(Text) (Touchable)
  DiaryAbstractList(List)
    DiaryAbstract(ListItem) (Touchable)
      DiaryMood(Image)
      SubView(View)
        DiaryTime(Text)
        DiaryTitle(Text)
```

接下来是阅读日记界面和写日记界面的组件结构。请读者先不要看下面的内容，自己在一张纸上对着前面的两张 UI 设计草图，写出这两个界面的组件结构。

阅读日记界面的组件结构是：

```
DiaryReader (View)
  FirstRow(View)
    ReturnButton(Text) (Touchable)
    PreviousDiaryButton(Text) (Touchable)
    NextDiaryButton(Text) (Touchable)
  SecondRow(View)
    DiaryMood(Image)
    SubView(View)
      DiaryTitle(Text)
      DiaryTime(Text)
    DiaryBody(TextInput)
```

在阅读日记界面的组件结构中，显示日记正文用的是 `TextInput` 组件而不是 `Text` 组件。`TextInput` 组件可以设置成不可编辑，这时它就是一个可以上下滚屏的 `Text` 组件。在我们还没有讨论可以滚屏显示的 `ScrollView` 组件时，我们先用它来替代 `ScrollView` 组件。

写日记界面的组件结构是：

```
DiaryWriter (View)
  FirstRow(View)
    ReturnButton(Text) (Touchable)
    SelectMoodButton(Text) (Touchable)
    SaveButton(Text) (Touchable)
  DiaryTitle(Text)
  DiaryBody(TextInput)
```

7.5.2.3 自下而上的设计

读者可能已经注意到了，这里描述的是自下而上的设计思路。从 UI 界面的底部（最基础的 React Native 组件）开始构建，每分出的一层组件构成了一个上层 React Native 组件，直到向上分层到一个界面的根 View 组件。

7.5.3 使用 React Native 组件搭建静态界面

得到了组件结构后，现在开始搭建 React Native 组件的静态界面。

日记列表静态界面的搭建需要用到第 8 章中介绍的 `ListView` 组件。为了在本章中就可以讨论 React Native 应用实现步骤，这里简化为在日记列表界面上只显示一个最新的日记“列表”。

这一步只编写各个界面对应组件的 `render` 函数，排列 `render` 函数中使用到的基础 React Native 组件，生成对应的样式。在编写中，开发者会不断地发现在某个界面中需要填入数据，这就是数据表。如果数据表比较多的话，则可以用一个文本文件记录下来。

在搭建界面前，请大家自行准备 5 张正方形 JPG 图片，分辨率在 100×100 以上，分别代表 5 种心情（如果想简单，则可以用纯白色、纯黑色、灰色等其他颜色图片替代）。将这些 JPG 文件放在项目目录的 `image` 子目录下。

我们先编写程序主框架和公用样式。

程序主框架见代码 7-16。

代码 7-16, `index.android.js` 或者 `index.ios.js`:

```
'use strict';
let React = require('react-native');
let {
  AppRegistry, StatusBarIOS, AsyncStorage
} = React;
let DiaryList=require('./DiaryList');
let DiaryWriter=require('./DiaryWriter');
let DiaryReader=require('./DiaryReader');
let Project19 = React.createClass({
  showDiaryList:function() {
    return (
      <DiaryList/>
    );
  },
  showDiaryReader:function() {
    return (
      <DiaryReader/>
    );
  },
  showDiaryWriter:function() {
    return (
      <DiaryWriter/>
    );
  },
  componentWillMount:function() {
    if ( StatusBarIOS != null ) StatusBarIOS.setHidden(true);    //隐藏 iOS 状态栏
                                                                //在 Android 平台上，StatusBarIOS 变量值为 null
  },
  render: function() {
    //return this.showDiaryList();
    //return this.showDiaryReader();
    return this.showDiaryWriter();
  }
});
```

```

    },
  });
AppRegistry.registerComponent('Project19', () => Project19);

```

在 Project19 组件的 render 函数中，三条语句分别用来显示三个界面，方便开发者实现静态界面。

因为各个界面可以用到一些公用的宽、高变量以及样式，所以将它们存放在公用样式文件中，以减少代码量，同时也让代码看起来简洁。公用样式文件名为 MCV.js。这是开发者自己定义的文件名，取 My Common Variable 三个单词的首字母。它的内容见代码 7-17。

代码 7-17, MVC.js:

```

'use strict';
let React = require('react-native');
let Dimensions = require('Dimensions');
let totalWidth = Dimensions.get('window').width;
let totalHeight = Dimensions.get('window').height;
let textSize = totalWidth / 18;
let readingUITitleHeight = textSize * 6;
let diaryBodyLine = totalHeight / textSize - 6 ;
let returnButtonHeight = textSize * 5;
let { StyleSheet } = React;
let MCV = StyleSheet.create({
  container: {      //container 是每个界面根 View 的样式
    top:2,           //这里 top 设置为 2，是为了与屏幕最上边拉开些距离。这种做法是有问题的。
    flex: 1,         //从截图或者手机运行界面来看，它导致了最下边的黑色边框没能显示出来。请大
                    //家仔细思考 top 键值与 flex:1 键值的关系。提示：设置了 flex:1, top 就不
                    //能设置其他值（默认值 0）。请大家自己动手修正这个 bug（不仅是简单地将 top
                    //设置为 0,修改时还需要保持按钮与屏幕最上边的距离,紧挨屏幕上边沿的按钮不美观）。
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#F5FCFF',
    borderColor: 'black', //给根 View 加黑色边框，是为了便于手机截图，清楚地
    borderWidth: 1       //显示界面
  },                 //如果把 border 两行去掉，上一个注释说的
                    //问题就发现不了了
  firstRow: {         //firstRow 用来定义屏幕最上方显示按钮的 View
    height:textSize *1.4 +2,
    flexDirection: 'row',

    width:totalWidth - 4,           // margin 设置为 2，两边加起来就是 4，所以宽度要减 4
    justifyContent: 'space-around', //使用这个键值，可以看到这一行里的三个按钮不论
    margin:2                       //屏幕宽度是多少都会乖乖地排列好，感觉还是很省心的
  },
  longButton: {        //定义一个比较长的 Text 做按钮
    height: textSize *1.4,
    backgroundColor: 'gray',
    width: textSize*10,
    borderRadius: 8,    //设置了圆角风格，在 iPhone 平台上显示正常，但在 Android
    textAlign:'center', //手机上没有圆角风格，所以下面的 Android 手机运行截图
    fontSize: textSize  //中的按钮都是直角的。在真正商用开发时，还是尽量让美工
  },                   //出图，用 Image 做按钮
  middleButton: {      //长度适中的 Text 按钮样式

```

```

        height:textSize *1.4 ,
        backgroundColor: 'gray',
        width:textSize*5,
        borderRadius: 8,
        textAlign:'center',
        fontSize: textSize
    },
    diaryAbstractList: {      //这是日记列表界面用来实现假列表的 view
        flex:1,
        margin:4,
        width:totalWidth-4,
        justifyContent: 'center',
        backgroundColor:'grey'
    },
    diaryBodyStyle: {        //显示、输入日记正文的 TextInput 组件样式
        flex:1,
        width:totalWidth-8, //读者可以试一试看不设置这个 width 的效果，并且注意在阅读
        fontSize:textSize, //日记和写日记界面中，日记显示（输入框）的左右两边能否与上//面的边对齐
        backgroundColor:'grey',
        margin:4
    },
    smallButton: {           //长度比较短的 Text 组件样式
        height:textSize *1.4 ,
        backgroundColor: 'gray',
        width:textSize*3,
        borderRadius: 8,
        textAlign:'center',
        fontSize: textSize
    },
    moodStyle: {             //显示记日记时心情的 Image 组件样式
        height:textSize*3.2,
        width:textSize*3.2,
        borderRadius:textSize*1.6
    },
    subViewInReader: {      //这个 View 用来在心情图片旁边显示日记的标题和时间
        width:totalWidth-5-textSize*3.2,
    },
    textInReader: {          //这是显示在心情图片旁边的标题和时间的 Text 组件样式
        height:textSize*1.4,
        fontSize: textSize,
        backgroundColor: 'grey',
        margin:2,
    },
    secondRow: {             //阅读日记界面用来展示心情图片、日记标题、日记时间的 View
        flexDirection: 'row',
        alignItems: 'center'
    },
    titleInputStyle: {       //写日记界面用来输入日记标题的 TextInput 组件样式
        fontSize: textSize,
        backgroundColor:'grey',
        height: textSize*2.4,
        color:'black',
        margin: 4,
    },

```



```

    borderWidth:2,
    borderColor: 'black'
  },
  searchBarTextInput: { //搜索关键字输入框样式定义
    backgroundColor: 'white',
    borderColor: 'black',
    borderWidth: 1,
    height:textSize *1.4 ,
    width: textSize*10,
    paddingTop: 0, //如果不设置,在 Android 平台 TextInput 组件内输入显示异常
    paddingBottom: 0, //如果不设置,在 Android 平台 TextInput 组件内输入显示异常
    top:1,
    left:1,
    fontSize: 14,
  }
});
module.exports=MCV;

```

其他界面只需要 `require('./MCV')` 就可以使用这个文件中定义的样式,而不需要重复定义某些样式。

日记列表静态界面搭建完成后,实现代码类似于代码 7-18。

在代码 7-18 中,使用三元运算符示范了视平台不同来定义不同的 React Native 组件。这个示例可以改为两个平台使用同一份 React Native 组件。这里为了演示这个知识点,还是分成了两份不同的 React Native 组件。

代码 7-18, DiaryList.js:

```

'use strict';
var React = require('react-native');
let angryMood = require('./image/angry.jpg') //为了测试界面先临时获取这张图片
let MCV = require('./MCV');
var {
  View, Text, TextInput, TouchableOpacity, Image, Platform
} = React;
let DiaryList = React.createClass({
  updataSearchKeyword:function(newWord) {
    //这个功能请读者自行实现
  },
  render: function() {
    return (
      <View style={MCV.container}>
        <View style={MCV.firstRow} > //下面的代码使用到了三元运算符,参见附录 A.8
          { (Platform.OS === 'android' )? //视平台不同显示不同的组件
            ( //Android 平台显示组件
              <View style={{borderWidth:1}}>
                <TextInput autoCapitalize="none"
                  placeholder='输入搜索关键字'
                  clearButtonMode="while-editing"
                  onChangeText={this.updataSearchKeyword}
                  style={MCV.searchBarTextInput}/>
              </View>
            ): ( //iOS 平台显示组件

```

```

        <TextInput autoCapitalize="none"
          placeholder=' 输入搜索关键字'
          clearButtonMode="while-editing"
          onChangeText={this.updateSearchKeyword}
          style={MCV.searchBarTextInput} />
      )
    } //视平台不同显示不同的组件结束
    <TouchableOpacity>
      <Text style={MCV.middleButton}>
        写日记
      </Text>
    </TouchableOpacity>
  </View>
  <View style={MCV.diaryAbstractList}>
    <View style={MCV.secondRow}>
      <Image style={MCV.moodStyle}
        source={angryMood} /> //这里应当是某变量，填入了值是为了测试界面
      <View style={MCV.subViewInReader}>
        <TouchableOpacity onPress={this.props.selectLististItem}>
          <Text style={MCV.textInReader}>
            某变量记录假日记列表标题
          </Text>
        </TouchableOpacity>
        <Text style={MCV.textInReader}>
          某变量记录假日记列表时间
        </Text>
      </View>
    </View>
  </View>
</View>
);
},
});
module.exports=DiaryList;

```

日记列表静态界面效果如图 7-4 所示。



图 7-4 日记列表静态界面效果

TextInput 组件在 Android 平台和 iOS 平台上的表现不同，为了让其在两个界面上显示一样。例程中使用了一些特殊的技巧。

阅读日记静态界面搭建完成后，实现代码类似于代码 7-19。

代码 7-19, DiaryReader.js:

```
'use strict';
var React = require('react-native');
let angryMood = require('./image/angry.jpg') //为了测试界面先临时获取这张图片
let MCV = require('./MCV');
var {
  View, Text, TextInput, TouchableOpacity, Image
} = React;
let DiaryReader = React.createClass({
  render: function() {
    return (
      <View style={MCV.container}>
        <View style={MCV.firstRow} >
          <TouchableOpacity>
            <Text style={MCV.middleButton}>
              返回
            </Text>
          </TouchableOpacity>
          <TouchableOpacity>
            <Text style={MCV.middleButton}>
              上一篇
            </Text>
          </TouchableOpacity>
          <TouchableOpacity>
            <Text style={MCV.middleButton}>
              下一篇
            </Text>
          </TouchableOpacity>
        </View>
        <View style={MCV.secondRow}>
          <Image style={MCV.moodStyle}
            source={angryMood} /> //这里应当是某变量，填入了值是为了测试界面
          <View style={MCV.subViewInReader}>
            <Text style={MCV.textInReader}>
              日记标题: 某变量
            </Text>
            <Text style={MCV.textInReader}>
              时间: 某变量
            </Text>
          </View>
        </View>
        <View>
          <TextInput style={[MCV.diaryBodyStyle, {color: 'black'}]}
            multiline={true}
            editable={false}
            value={'某变量记录日记正文'} />
        </View>
      </View>
    );
  },
});
module.exports=DiaryReader;
```

阅读日记静态界面效果如图 7-5 所示。



图 7-5 阅读日记静态界面效果

在代码 7-19 中，我们通过 `TextInput` 组件，而不是 `Text` 组件来显示文本正文。这是因为用户输入的日记有可能会超过日记正文屏幕显示空间，而此时还没有介绍 `ScrollView` 组件（它可以提供可上下滚动的显示空间），因此通过 `TextInput` 组件来达到类似的屏幕滚动效果。

在上面的代码中，最后一个 `TextInput` 组件，因为设置了 `editable={false}` 属性让 `TextInput` 组件中的文本不可编辑，`React Native` 会自动让其中的文本颜色变淡。我们通过强行设置颜色样式 `{color: 'black'}` 来矫正这个行为，让文本颜色方便阅读。

写日记静态界面搭建完成后，实现代码类似于代码 7-20。

代码 7-20, DiaryWriter.sj:

```
'use strict';
var React = require('react-native');
let MCV = require('./MCV');
var {
  View, Text, TextInput, TouchableOpacity
} = React;
let DiaryWriter = React.createClass({
  returnPressed:function() {
    Alert.alert(
      '请确认',
      '你确定要退回日记列表吗？',
      [
        {text: '确定' },
        {text: '取消' }
      ]
    );
  }
});
```

```

    },
    render: function() {
      return (
        <View style={MCV.container}>
          <View style={MCV.firstRow} >
            <TouchableOpacity onPress={this.returnPressed}>
              <Text style={MCV.smallButton}>
                返回
              </Text>
            </TouchableOpacity>
            <TouchableOpacity onPress={this.selectMood}>
              <Text style={MCV.longButton}>
                某变量当前按钮文字
              </Text>
            </TouchableOpacity>
            <TouchableOpacity>
              <Text style={MCV.smallButton}>
                保存
              </Text>
            </TouchableOpacity>
          </View>
          <TextInput style={MCV.titleInputStyle}
            placeholder={'写个日记标题吧'}/>
          <TextInput style={MCV.diaryBodyStyle}
            multiline={true}
            placeholder={'日记正文请在此输入'}/>
        </View>
      );
    },
  });
module.exports=DiaryWriter;

```

写日记静态界面效果如图 7-6 所示。



图 7-6 写日记静态界面效果

在实现这三个界面的静态代码中，出现“某变量”的地方就是“需要显示的数据表”中的一项。我们将在 7.5.4.2 节中确定每个变量的拥有者，然后对其定性、命名。

在实现这三个界面的静态代码中，还有很多可触摸基本组件，每一个可触摸基本组件都至少对应一个需要处理的事件，这些事件构成了一个事件列表。我们将在 7.5.4.3 节中确定这些事件，为其分配处理者。

7.5.4 React Native 组件分层

静态界面搭建完成后，开发者需要对 React Native 组件进行分层。不同于基础组件的分层，React Native 组件分层是区分控制 React Native 组件与显示 React Native 组件的（当然也可能有两个功能都有的组件）。

7.5.4.1 分析界面之间的联系

如果业务逻辑需要从界面 A 跳转到界面 B，或者需要把界面 A 呈现时得到（用户输入或者网络获取等）的数据展示在界面 B 上，我们就说界面 A 需要给界面 B 发送消息。如果界面 A 需要给界面 B 发送消息，但界面 B 不需要给界面 A 发送消息，那么在 React Native 应用设计中，界面 B 对应的 React Native 组件应当成为界面 A 对应的 React Native 组件的子组件。如果界面 A、B 都需要给对方发送消息，那么它们对应的 React Native 组件就应当有一个共同的父控制组件。

按这个规则分析，日记例程中的三个界面对应的 React Native 组件应当有一个共同的父控制组件。Project19 组件做父控制组件正好合适。

本例程简单，两层结构就可以了。在其他项目开发中，开发者应当按照上述逻辑，需要多少层就设计多少层。

显示组件与控制组件的分层也是按照自下而上的原则进行的。

7.5.4.2 确定数据的显示者与拥有者

这一步需要确定哪个组件会显示哪些数据，哪个组件会改变或者说拥有哪些数据。

在 React Native 框架中，数据是沿着组件树从上到下单向流动的。拥有数据的 React Native 组件并不一定负责显示该数据，它经常把自己拥有的数据（存储在它的状态机变量中）作为一个子组件的属性传递给子组件，由子组件来显示它。

在确定数据的显示者与拥有者时，我们同样遵循从下到上的原则。

对 7.5.3 节生成的数据表中的每一个数据，找出负责在手机界面上渲染数据的组件，然后判断该数据是否可以被该组件的上层拥有，如果可以（意味着这个数据不会在本组件中被改变），把它丢给上层组件（意味着这个数据由上层组件通过 props 传递下来）；如果不可以，那么这个数据可能会是本组件的一个状态机变量。继续将这个数据向上层丢的过程，直到数据被丢到了最上层或者无法向上丢了。当这一步完成后，对于每一个组件，开发者就得到了该组件需要使用的属性

表，以及该组件可能拥有的状态机变量表。

7.5.4.3 确定状态机变量

确定状态机变量的方法在 2.5.4 节中已经详细描述了，此处不再赘述。

对 7.5.4.2 节确定的每个组件可能拥有的状态机变量表按这个方法分析，就可以得出每个组件的状态机变量的最小集。

7.5.4.4 确定各事件的接收者与处理者

对 7.5.3 节得到的事件列表中的每一个事件，确定其处理者。

如果事件是在上层处理的，那么把事件丢给上层组件（意味着上层需要提供给事件接收层一个回调函数，并且把这个回调函数作为属性传给负责接收事件的组件）。在上层组件中写出这个回调函数的原型，并且将它传递到下层组件，下层组件再将它与按钮的按下事件挂上钩。

如果事件是由本层组件处理的，那么写出这个处理函数的原型。

继续这个过程，直到每个事件都被丢到了负责处理它的组件，并且实现了该事件的处理函数原型。

7.5.4.5 非界面触发事件的接收者与处理者

非界面触发事件，是指不能由 React Native 基础组件（React Native 框架提供的组件称为 React Native 基础组件）上报的事件。比如各种监听事件（Android 手机后退键监听、混合开发原生代码侧消息监听等）、定时器事件等。

首先需要确定这些事件的接收者在 React Native 组件分层结构中的位置。思路是：观察分析某个事件会造成哪些 React Native 组件（每个组件代表一个界面）被改变，将这些 React Native 组件看成一个树形结构（可能需要补充某些控制 React Native 组件），将接收者确定为这个树型结构的最高节点，然后让这个事件被触发的消息按业务逻辑处理，在这个树形结构中通过属性从上向下流动。

本例程没有非界面触发事件，但作为 React Native 应用实现步骤之一，这一点还是需要说清楚。

7.5.5 实现各组件业务逻辑

各组件的属性、状态机变量定型后，就可以实现各组件业务逻辑了。在实现业务逻辑的过程中，有可能需要给某组件增加成员变量、成员方法等。业务逻辑包括但不限于：

1. 直接渲染状态机变量和属性

在各组件的 render 函数的相应位置填入状态机变量和属性名称。

2. 将状态机变量以属性的方式交给下层组件

很多组件的状态机变量是由其子组件负责显示的，需要通过属性将状态机变量传给子组件。这个属性的名称在 7.5.4.2 节得到的属性表中已经被开发者确定。在 `render` 函数的子组件声明中添加对属性的赋值语句，实现状态机变量的传递。

3. 实现状态机变量的其他业务控制逻辑

某些状态机变量是用来进行业务逻辑控制的。比如在日记例程的 `Project19` 组件中，声明了 `uiCode` 状态机变量来控制当前显示哪一个界面。那么在 `Project19` 组件的 `render` 函数中，实现就是：

```
render: function() {
    if ( StatusBarIOS != null ) StatusBarIOS.setHidden(true);
    if (this.state.uiCode === 1 ) return this.testDiaryList();
    if (this.state.uiCode === 2 ) return this.testDiaryReader();
    if (this.state.uiCode === 3 ) return this.testDiaryWriter();
},
```

7.5.6 日记例程（上）总结

在日记例程上部分中，为了方便读者专注于本节讨论的 React Native 应用实现步骤，我们暂时不实现搜索功能，也就是暂时不去写搜索键被按下的处理逻辑。

7.5.6.1 无直接关系的 React Native 组件间消息传递

在总结前，还需要单独用一小节来强调如何在无直接关系的 React Native 组件间实现消息传递。虽然在前面已经介绍了相关内容，但很多读者可能还没意识到它的性质及重要性。

很多 React Native 初学者都发现自己有在无直接关系的 React Native 组件间实现消息传递的需求，但找不到实现方法。实现方法之一就是通过对无直接关系的公共父组件来间接传递消息。对于这种间接传递消息，消息发送者不需要指定消息接收者，由公共父组件按业务逻辑来判定消息接收者是谁。

在日记例程中，`DiaryWriter` 就需要把用户刚写的日记标题、心情传递给 `DiaryList` 组件显示在日记列表界面上。它们之间没有直接关系，彼此都不知道对方的存在，但通过一个公共的控制组件实现了这一点。

这个例程中的实现虽然简单，但却有效，请读者仔细回思整个实现的步骤。

7.5.6.2 日记例程（上）最终代码

经过 7.5.4 节讨论的各步骤，7.5.5 节最终实现的日记例程的代码见代码 7-21 至代码 7-24。`MCV.js` 已经在 7-17 中给出，因为这部分代码是样式设置部分，不需要再有任何改动，因此在这里不再给出。

代码 7-21，`index.android.js` 或者 `index.ios.js`：

```
'use strict';
```



```

let React = require('react-native');
let {
  AppRegistry, StatusBarIOS, AsyncStorage
} = React;
let DiaryList = require('./DiaryList');
let DiaryWriter = require('./DiaryWriter');
let DiaryReader = require('./DiaryReader');
let angryMood = require('./image/angry.jpg');
let peaceMood = require('./image/peace.jpg');
let happyMood = require('./image/happy.jpg');
let sadMood = require('./image/sad.jpg');
let miseryMood = require('./image/misery.jpg');
let Project19 = React.createClass({
  realDairyList: null, //因为在日记例程（上）中没有实现真正的日记列表
                        //日记列表保存在 Project19 的成员变量中，而不是状态机变量中
  listIndex: 0, //另一个成员变量，指示当前假日记列表上列出的记录是列表的第几条
  bubbleSortDiaryList: function() { //因为 AsyncStorage API 不能保证读取的顺序
    //使用冒泡排序对日记列表进行排序
    //选择冒泡排序是因为它的算法简单

    let tempObj;
    let len1 = this.realDairyList.length; //对需要在循环中访问的数据缩短访问时间
    let len2 = len1 - 1; //对需要在循环中访问的数据缩短访问时间
    for (let i = 0; i < len1; i++) {
      for (let j = 0; j < len2 - i; j++) {
        if (this.realDairyList[j].index > this.realDairyList[j + 1].index) {
          tempObj = this.realDairyList[j];
          this.realDairyList[j] = this.realDairyList[j + 1];
          this.realDairyList[j + 1] = tempObj;
        }
      }
    }
  }, // bubbleSortDiaryList 函数定义结束
  getInitialState: function() {
    AsyncStorage.getAllKeys().then( //获取存储中所有的 key
      (keys) => {
        AsyncStorage.multiGet(keys).then(//通过 keys 获取存储中所有的数据
          (results) => {
            let j = results.length;
            this.realDairyList = Array();
            for (let i = 0; i < j; i++) {
              //取得数据并利用 JSON 类的 parse 方法生成对象，插入日记列表
              this.realDairyList[i] = JSON.parse(results[i][1]);
            }
            this.bubbleSortDiaryList(); //对日记列表进行冒泡排序
            if (j > 0) { //日记列表中有数据，取出最后一条数据
              j--;
              this.listIndex = j;
              let newMoodIcon;
              switch (this.realDairyList[j].mood) {
                case 2:
                  newMoodIcon = angryMood;
                  break;
                case 3:
                  newMoodIcon = sadMood;

```

```

        break;
      case 4:
        newMoodIcon = happyMood;
        break;
      case 5:
        newMoodIcon = miseryMood;
        break;
      default:
        newMoodIcon = peaceMood;
    }
    let newtitle = this.realDairyList[j].title;
    let newbody = this.realDairyList[j].body;
    //利用 Date 的构造函数, 从字符串中得到 Date 类型数据
    let ctime = new Date(this.realDairyList[j].time);
    let timeString = '' + ctime.getFullYear() + ' 年 ' +
(ctime.getMonth() + 1) + ' 月 ' + ctime.getDate() + ' 日 ' + ' 星期 ' + (ctime.getDay() + 1) + ' ' +
' + ctime.getHours() + ':' + ctime.getMinutes();
    this.setState(() => { //将最后一条数据写入状态机变量
      return {
        diaryMood: newMoodIcon,
        diaryTime: timeString,
        diaryTitle: newtitle,
        diaryBody: newbody
      };
    });
  } else { //日记列表中没有数据
    this.setState(() => {
      return {
        diaryTime: '没有历史日记',
        diaryTitle: '没有历史日记',
        diaryBody: ''
      };
    });
  }
}
).catch((error) => {
  console.log(' then error:' + error);
}); // AsyncStorage.multiGet(keys).then 语句结束
}, (error) => {
  console.log(' getAllKeys error:' + error.message);
}
); // AsyncStorage.getAllKeys().then 语句结束
return { //上面的读取数据操作是异步操作, 不会马上得到数据, 在这里给
  //状态机变量赋初值
  uiCode: 1,
  diaryMood: peaceMood,
  diaryTime: '读取中.....',
  diaryTitle: '读取中.....',
  diaryBody: '读取中.....'
};
}, // getInitialState 函数定义结束
readingPreviousPressed: function() { //阅读日记界面请求读上一篇日记的处理函数
  if (this.listIndex === 0) return; //已经显示的是最上一篇日记
  this.listIndex--;

```

```

let j = this.listIndex;
let newMoodIcon;
switch (this.realDairyList[j].mood) {           //准备上一篇日记的心情图片
  case 2:
    newMoodIcon = angryMood;
    break;
  case 3:
    newMoodIcon = sadMood;
    break;
  case 4:
    newMoodIcon = happyMood;
    break;
  case 5:
    newMoodIcon = miseryMood;
    break;
  default:
    newMoodIcon = peaceMood;
}
let newtitle = this.realDairyList[j].title;
let newbody = this.realDairyList[j].body;
let ctime = new Date(this.realDairyList[j].time);
let timeString = '' + ctime.getFullYear() + '年' + (ctime.getMonth() + 1) + '
月' + ctime.getDate() + '日 星期' + (ctime.getDay() + 1) + ' ' + ctime.getHours() + ':' +
+ ctime.getMinutes();
this.setState(() => {           //将上一篇日记的相关数据放入状态机变量
  return {
    diaryMood: newMoodIcon,
    diaryTime: timeString,
    diaryTitle: newtitle,
    diaryBody: newbody
  };
});
},
readingNextPressed: function() {           //阅读日记界面请求读下一篇日记的处理函数
  if ( this.realDairyList.length === 0 ) return;
  if (this.listIndex === (this.realDairyList.length - 1)) return; //已经到最后一篇日记
  this.listIndex++;
  let j = this.listIndex;
  let newMoodIcon;
  switch (this.realDairyList[j].mood) {
    case 2:
      newMoodIcon = angryMood;
      break;
    case 3:
      newMoodIcon = sadMood;
      break;
    case 4:
      newMoodIcon = happyMood;
      break;
    case 5:
      newMoodIcon = miseryMood;
      break;
    default:
      newMoodIcon = peaceMood;
  }
}

```

```

        let newtitle = this.realDairyList[j].title;
        let newbody = this.realDairyList[j].body;
        let ctime = new Date(this.realDairyList[j].time);
        let timeString = '' + ctime.getFullYear() + '年' + (ctime.getMonth() + 1) + '
月' + ctime.getDate() + '日 星期' + (ctime.getDay() + 1) + ' ' + ctime.getHours() + ':' +
+ ctime.getMinutes();
        this.setState(() => {                                //将下一篇日记的相关数据放入状态机变量
            return {
                diaryMood: newMoodIcon,
                diaryTime: timeString,
                diaryTitle: newtitle,
                diaryBody: newbody
            };
        });
    },
    returnPressed: function() {                                //阅读日记界面、写日记界面返回日记列表界面的处理函数
        this.setState(() => {
            return {
                uiCode: 1
            }
        });
    },
    //写日记界面保存日记并返回日记列表界面的处理函数
    saveDiaryAndReturn: function(newDiaryMood, newDiaryBody, newDiaryTitile) {
        let cTime = new Date();                                //获取当前时间
        let timeString = '' + cTime.getFullYear() + '年' + (cTime.getMonth() + 1) + '月'
+ cTime.getDate() + '日 星期' + (cTime.getDay() + 1) + ' ' + cTime.getHours() + ':' +
cTime.getMinutes();
        let aDiary = Object();
        aDiary.title = newDiaryTitile;
        aDiary.body = newDiaryBody;
        aDiary.mood = newDiaryMood;
        aDiary.time = cTime;
        aDiary.sectionID = '' + cTime.getFullYear() + '年' + (cTime.getMonth() + 1) + '
月'; //sectionID 用来对日记列表进行分段显示（见第8章）
        aDiary.index = Date.parse(cTime); //从当前时间生成唯一值，用来索引日记列表
        //这个值精确到毫秒，可以认为它是唯一的
        AsyncStorage.setItem('' + aDiary.index, JSON.stringify(aDiary)).then(
            () => { //将新的日记存储在本地存储中
                console.log('Saving succeed');
            }, (error) => {
                console.log('Saving failed, error' + error.message);
            }
        );
    },
    let totalLength = this.realDairyList.length;
    this.realDairyList[totalLength] = aDiary; //将新的日记加入内存中的日记列表
    this.listIndex = totalLength;
    let newMoodIcon;
    switch (newDiaryMood) {
        case 2:
            newMoodIcon = angryMood;
            break;
        case 3:
            newMoodIcon = sadMood;
    }

```

```

        break;
    case 4:
        newMoodIcon = happyMood;
        break;
    case 5:
        newMoodIcon = miseryMood;
        break;
    default:
        newMoodIcon = peaceMood;
    }
    this.setState(() => {          //更新状态机变量
        return {
            uiCode: 1,
            diaryTime: timeString,
            diaryTitle: newDiaryTitile,
            diaryMood: newMoodIcon,
            diaryBody: newDiaryBody
        };
    });
},
writeDiary: function() {        //写日记按钮被按下时的处理函数
    this.setState(() => {
        return {
            uiCode: 3
        };
    });
},
searchKeyword: function(keyword) {    //搜索按钮被按下时的处理函数，仅实现原型
    console.log('search button pressed, the keyword is:' + keyword);
},
selectLististItem: function() {        //日记列表中某条记录被选中时的处理函数
    this.setState(() => {
        return {
            uiCode: 2
        };
    });
},
showDiaryList: function() {
    return (          //注意，如何将状态机常量作为属性向下层 React Native 组件传递
        //注意，如何将上层组件的某些函数作为回调函数利用属性向下层传递
        <DiaryList fakeListTitle={this.state.diaryTitle}
            fakeListTime={this.state.diaryTime}
            fakeListMood={this.state.diaryMood}
            selectLististItem={this.selectLististItem}
            searchKeyword={this.searchKeyword}
            writeDiary={this.writeDiary}/>
    );
},
showDiaryWriter: function() {
    return (
        <DiaryWriter
            returnPressed={this.returnPressed}
            saveDiary={this.saveDiaryAndReturn}/>
    );
},
},

```

```

showDiaryReader: function() {
  return (
    //注意,如何将状态机常量作为属性向下层 React Native 组件传递
    //注意,如何将上层组件的某些函数作为回调函数利用属性向下层传递
    <DiaryReader returnToDiaryList={this.retrunToDiaryList}
      diaryTitile={this.state.diaryTitle}
      diaryMood={this.state.diaryMood}
      diaryTime={this.state.diaryTime}
      readingPreviousPressed={this.readingPreviousPressed}
      returnPressed={this.returnPressed}
      readingNextPressed={this.readingNextPressed}
      diaryBody={this.state.diaryBody}/>
  );
},
componentWillMount: function() {
  if (StatusBarIOS !== null) StatusBarIOS.setHidden(true);
},
render: function() {
  if (this.state.uiCode === 1) return this.showDiaryList();
  if (this.state.uiCode === 2) return this.showDiaryReader();
  if (this.state.uiCode === 3) return this.showDiaryWriter();
},
});
AppRegistry.registerComponent('Project19', () => Project19);

```

代码 7-22, DiaryList.js:

```

'use strict';
var React = require('react-native');
let angryMood = require('./image/angry.jpg') //为了测试界面先临时获取这张图片
let MCV = require('./MCV');
var {
  View, Text, TextInput, TouchableOpacity, Image, Platform
} = React;
let DiaryList = React.createClass({
  let DiaryList = React.createClass({
    keyword:'',
    updateSearchKeyword:function(newWord) {
      this.keyword=newWord;
      //将用户输入的搜索关键字交给上层组件,由上层组件对日记列表进行处理,只显示日记
      //标题中包含关键字的日记
    },
    render: function() {
      .....
      <TouchableOpacity onPress={this.props.writeDiary}> //调用回调函数
        <Text style={MCV.middleButton}>
          写日记
        </Text>
      </TouchableOpacity>
      .....//下面的代码与代码 7-18 没有区别,不再列出
    }
  });

```

代码 7-23, DiaryReader.js:

```

var React = require('react-native');
let MCV = require('./MCV');
var {
  View, Text, TextInput, TouchableOpacity, Image

```

```

} = React;
let DiaryReader = React.createClass({
  render: function() {
    return (
      <View style={MCV.container}>
        <View style={MCV.firstRow} >
          <TouchableOpacity onPress={this.props.returnPressed}>
            <Text style={MCV.middleButton}>
              返回
            </Text>
          </TouchableOpacity>
          <TouchableOpacity onPress={this.props.readingPreviousPressed}>
            <Text style={MCV.middleButton}>
              上一篇
            </Text>
          </TouchableOpacity>
          <TouchableOpacity onPress={this.props.readingNextPressed}>
            <Text style={MCV.middleButton}>
              下一篇
            </Text>
          </TouchableOpacity>
        </View>
        <View style={MCV.secondRow}>
          <Image style={MCV.moodStyle}
            source={this.props.diaryMood}/> //上层传入的属性被渲染
          <View style={MCV.subViewInReader}>
            <Text style={MCV.textInReader}>
              标题: {this.props.diaryTitile} //上层传入的属性被渲染
            </Text>
            <Text style={MCV.textInReader}>
              时间: {this.props.diaryTime} //上层传入的属性被渲染
            </Text>
          </View>
        </View>
        <TextInput style={[MCV.diaryBodyStyle,{color: 'black'}]}
          multiline={true}
          editable={false}
          value={this.props.diaryBody}/> //上层传入的属性被渲染
      </View>
    );
  },
});
module.exports=DiaryReader;

```

代码 7-24, DiaryWriter.js:

```

var React = require('react-native');
let MCV = require('./MCV');
var {
  View, Text, TextInput, TouchableOpacity, Alert
} = React;
let DiaryWriter = React.createClass({
  diaryTitle: 'test title',
  diaryBody: 'test body',
  moodCode: 0,
  getInitialState: function() {

```

```

    return {
      moodText: '请选择心情'
    };
  },
  returnPressed: function() {
    Alert.alert(
      '请确认',
      '你确定要退回日记列表吗? ',
      [
        //上层传入的回调函数被调用
        {text: '确定', onPress: this.props.returnPressed },
        {text: '取消' }
      ]
    );
  },
  selectMood: function() {
    let tempString;
    if ( this.moodCode === 5 ) this.moodCode = 1;
    else this.moodCode = this.moodCode+1;
    switch(this.moodCode) {
      case 1:
        tempString = '现在的心情: 平静';
        break;
      case 2:
        tempString = '现在的心情: 愤怒';
        break;
      case 3:
        tempString = '现在的心情: 悲伤';
        break;
      case 4:
        tempString = '现在的心情: 高兴';
        break;
      case 5:
        tempString = '现在的心情: 痛苦';
        break;
    }
    this.setState( ()=>{
      return {
        moodText: tempString
      };
    });
  },
  render: function() {
    return (
      <View style={MCV.container}>
        <View style={MCV.firstRow} >
          <TouchableOpacity onPress={this.returnPressed}>
            <Text style={MCV.smallButton}>
              返回
            </Text>
          </TouchableOpacity>
          <TouchableOpacity onPress={this.selectMood}>
            <Text style={MCV.longButton}>
              {this.state.moodText}
            </Text>
          </TouchableOpacity>
        </View>
      </View>
    );
  }
};

```



```

        </TouchableOpacity> //上层传入的属性被渲染
        <TouchableOpacity onPress={()=>this.props.saveDiary(this.moodCode,
            this.diaryBody,this.diaryTitle)}>
            <Text style={MCV.smallButton}>
                保 存
            </Text>
        </TouchableOpacity>
    </View>
    <TextInput style={MCV.titleInputStyle}
        placeholder={'写个日记标题吧'}
        onChangeText={(text)=>{this.diaryTitle=text}}/>

    <TextInput style={MCV.diaryBodyStyle}
        multiline={true}
        placeholder={'日记正文请在此输入'}
        onChangeText={(text)=>this.diaryBody=text}/>
</View>
    );
    },
    });
module.exports=DiaryWriter;

```

7.5.6.3 下一步工作

现在日记例程完成了一半，要真正实现一个日记例程，还需要使用 React Native 框架的 ListView 组件。我们将在第8章中完成整个日记例程。

完成日记例程还需要能从存储中读取以前保存的日记数据，我们也在第8章中完成这个工作。

第 8 章

ScrollView 和 ListView

ScrollView 组件允许用户左、右或者上、下滑动来查看原来显示在屏幕外的内容。

ListView 组件用来呈现一个列表，也可以通过上、下滑动来展示原来显示在屏幕外的内容。

对这两个组件大家都很熟悉，不再过多介绍它们的 UI 特性。

8.1 ScrollView 组件

ScrollView 组件封装了两大操作系统平台提供的滚动视图功能，并将滚动视图功能与触摸响应系统集成起来。

ScrollView 组件支持 View 组件的所有属性，也就是支持 View 组件的所有样式设置。ScrollView 组件没有自己独有的样式设置

8.1.1 ScrollView 组件属性

ScrollView 组件除了具有 View 组件的所有属性外，还有如下自己独有的属性。

- `contentContainerStyle` 是样式对象类型的属性，用来定义 ScrollView 组件的容器（ScrollView 组件的所有子组件都在此容器中展示）样式。
- `horizontal` 是布尔类型的属性。当它为 `true` 时，ScrollView 的所有子组件将会水平排列；当它为 `false` 时，ScrollView 组件的所有子组件将垂直排列。它的默认值是 `false`。
- `KeyboardDismissMode` 是字符串类型的属性。它的取值可以为 `none`、`interactive`、`on-drag`。它决定了 ScrollView 组件中某子组件调出软键盘后，是否允许通过拉软键盘这个手势让软键盘消失。`none` 是不允许；`on-drag` 表示在手势开始时，软键盘会消失；`interactive` 表示键盘消失的动画会与手势进展交互对应，如果用户向上回拉，软键盘不会消失。Android 操作系统只支持 `none` 取值。
- `keyboardShouldPersistTaps` 是布尔类型的属性。当它为 `false` 时，在文本输入框外触按屏幕将会使软键盘消失；为 `true` 时不会。它的默认值是 `false`。
- `onContentSizeChange` 是一个回调函数，当 ScrollView 组件的容器 View 宽、高被改变时，这个回调函数将被执行。
- `onScroll` 是一个回调函数，当 ScrollView 组件被滑动时，每一帧的画面改变都会触发一次

此函数。可以通过设置 `scrollEventThrottle` 属性来控制回调的频率。

- `scrollEventThrottle` 是数值类型的属性，它只对 iOS 平台有效。
- `removeClippedSubviews` 是布尔类型的属性。它的默认值为 `true`。当它为 `true` 时，React Native 框架通过使不在屏幕范围内的子 View 暂不处理计算而提高滑动效果体验。当 ScrollView 中的内容很长时，这个设置很有用。但这个方法还在实验阶段，如果开发者发现 ScrollView 组件运行有问题，则可以尝试将此属性设为 `false`。
- `showsHorizontalScrollIndicator` 是布尔类型的属性。当它为 `true` 时，水平方向的 ScrollView 组件会有一个滑动指示器（类似于窗口滚卷条的 UI 部件）。
- `showsVerticalScrollIndicator` 是布尔类型的属性。当它为 `true` 时，垂直方向的 ScrollView 组件会有一个滑动指示器。

8.1.2 ScrollView 组件 iOS 平台专有属性

- `alwaysBounceHorizontal` 是布尔类型的属性。当 `horizontal` 属性的值为 `true` 时，它的默认值为 `true`，否则为 `false`。当它为 `true` 时，ScrollView 在手指滑动到尽头时会有一个弹动效果，即使此时 ScrollView 的内容宽度小于 ScrollView 的宽度。
- `alwaysBounceVertical` 属性的作用与 `alwaysBounceHorizontal` 类似，只是它是用来控制垂直方向滑动效果的。当 `horizontal` 属性为 `false` 时，它的默认值为 `true`，否则为 `false`。
- `automaticallyAdjustContentInsets` 是布尔类型的属性。它决定了当 ScrollView 被放置在导航栏、分页栏、工具栏后时，iOS 系统是否自动调整它的内容。它的默认值是 `true`。
- `bounces` 是布尔类型的属性。当它为 `true` 时，如果 ScrollView 显示内容的宽（或者高）大于 ScrollView 的宽（或者高）时，在滑动到尽头时会有一个弹动效果。当它为 `false` 时，则不会有这个效果，即使设置了 iOS 原生代码中的 `alwaysBounce*` 值为 `true`。
- `bouncesZoom` 是布尔类型的属性。当它为 `true` 时，使用手势操作可以使 ScrollView 缩放超过它设定的最大/最小值，当手势操作结束后，ScrollView 会回到它允许的最大/最小值。当它为 `false` 时，不允许超过最大/最小值。
- `centerContent` 是布尔类型的属性。它的默认值是 `false`。当它为 `true` 时，如果 ScrollView 显示内容的宽、高小于 ScrollView 的宽、高，ScrollView 会自动将内容居中显示；如果显示内容的宽、高大于 ScrollView 的宽、高，则不会有任何效果。
- `contentInset` 是类类型的属性。它接收定义为 `{top: number, left: number, bottom: number, right: number}` 的对象。它定义了 ScrollView 相对父 View 边框从哪里开始显示。默认值为 `{0, 0, 0, 0}`。
- `contentOffset` 是 `PointPropType` 类型的属性，用来手动设置开始滑动的偏移值。它的默认值是 `{x: 0, y: 0}`。
- `decelerationRate` 是数值类型的属性。它是一个浮点数值，决定了当手指离开屏幕时，ScrollView 的滑动减速效果有多快。合理的设置值区间为：0.998（正常速度）~ 0.9（快速减速）。
- `directionalLockEnabled` 是布尔类型的属性。它的默认值为 `false`。当它为 `true` 时，ScrollView

会尝试只允许垂直（或者水平）方向滑动。

- `maximumZoomScale` 是数值类型的属性,用来设置允许最大的缩放比例。它的默认值为 1.0。
- `minimumZoomScale` 是数值类型的属性,用来设置允许最小的缩放比例。它的默认值为 1.0。
- `zoomScale number` 是数值类型的属性,用来设置当前 `ScrollView` 组件的缩放比例。它的默认值为 1.0。
- `onScrollAnimationEnd` 是一个回调函数,在滑动动画效果结束时将被调用。
- `pagingEnabled` 是布尔类型的属性。它的默认值是 `false`。当它为 `true` 时, `ScrollView` 组件中显示的内容在滑过整数倍的 `ScrollView` 宽、高时会停止滑动。这个特性可用来实现水平方向整页滑动。
- `scrollEnabled` 是布尔类型的属性。它的默认值是 `true`。当它为 `false` 时,不允许 `ScrollView` 中的内容滑动。
- `scrollEventThrottle` 是数值类型的属性,用来控制滑动事件的触发频率(每秒触发多少次)。这个数值设得越高,追踪 `ScrollView` 组件位置的代码准确度就越高。但这会导致滑动体验变差。它的默认值是 0,表示滑动事件只在 `ScrollView` 被滑动时触发一次。
- `scrollIndicatorInsets` 是类类型的属性。它接收定义为`{top: number, left: number, bottom: number, right: number}`的对象。它定义了 `ScrollView` 滑动指示器相对父 `View` 边框从哪里开始显示。它的默认值为`{0, 0, 0, 0}`。
- `scrollsToTop` 是布尔类型的属性。它的默认值是 `true`。当它为 `true` 时,用户点击状态栏将使 `ScrollView` 回到顶部。
- `SnapToInterval` 是数值类型的属性。当这个值被设定时, `ScrollView` 每滑过整数倍的 `SnapToInterval` 宽、高时会停止滑动。设置它可以给宽（或者高）小于 `ScrollView` 宽（或者高）的子组件提供类似于分页的效果。它需要配合 `snapToAlignment` 属性使用。
- `snapToAlignment` 是字符串类型的属性,它的取值可以是 `start`、`center`、`end`。当 `SnapToInterval` 被设定时, `snapToAlignment` 定义停止滑动的行为。`start` 使分页的头部与 `ScrollView` 的头部对齐, `center` 是中部对齐, `end` 是底部对齐。
- `stickyHeaderIndices` 是数值数组类型的属性。这个数组将决定哪个子组件会固定在屏幕上方而不会在滑动时滑出屏幕。比如 `stickyHeaderIndices=[0]`将使第一个子组件固定在 `ScrollView` 的顶部。这个属性不能与 `horizontal={true}`同时使用。
- `onRefreshStart` 是 iOS 平台特有的回调函数。当这个回调函数被提供时,会显示一个 `UIRefreshControl` 控件,并且在 `UIRefreshControl` 控件被激活时调用回调函数。

8.1.3 ScrollView 组件 Android 平台专有属性

- `onScrollBeginDrag` 是函数类型的属性。开发者可以通过它指定一个回调函数,当开始用手指拖动 `ScrollView` 组件时,这个回调函数会被调用。
- `onScrollEndDrag` 是函数类型的属性。开发者可以通过它指定一个回调函数,当结束用手指拖动 `ScrollView` 组件时,这个回调函数会被调用。
- `sendMomentumEvents` 是布尔类型的属性。它的默认值是 `true`。当它为 `true` 时, `ScrollView`

滚动趋势事件会被上报。

提示：滚动趋势是指使用手指快速地划过屏幕，这时 ScrollView 组件会顺着手指的方向继续滚动一段时间，不会在用户手指离开屏幕后马上停止滚动。

- `onMomentumScrollBegin` 是函数类型的属性。开发者可以通过它指定一个回调函数，当 ScrollView 组件的滚动趋势开始时，这个回调函数会被调用。
- `onMomentumScrollEnd` 是函数类型的属性。开发者可以通过它指定一个回调函数，当 ScrollView 组件的滚动趋势结束时，这个回调函数会被调用。

8.1.4 ScrollView 组件的公开成员函数

除了大部分组件都有的公开成员函数 `setNativeProps` 和 `measure` 函数外，ScrollView 组件还提供了 `scrollTo` 函数，以让当前的 ScrollView 快速地定位到指定屏幕位置。

`scrollTo` 函数要求提供一个对象作为调用它的参数。这个对象的数据结构是：

```
{
  x: aNumber,           // 欲定位位置的横坐标
  y: aNumber,           // 欲定位位置的纵坐标
  animated: aBool       // 定位到指定位置时需要动画效果还是一下子跳过去
}
```

假设已经得到一个 ScrollView 组件的引用为 `aScrollViewRef`，使用这个成员函数的示例代码如下（需要运行在 React Native 0.20.0 版本或更高版本中）：

```
.....
aScrollViewRef.scrollTo({x: 0, y: 50, animated: true})
.....
```

8.1.5 RefreshControl 组件

RefreshControl 是专门为 ScrollView 组件服务的组件。当 ScrollView 被拉到顶部（y:0）时，如果给 ScrollView 的 `refreshControl` 属性赋值一个 RefreshControl 组件，则会显示这个 RefreshControl 组件。开发者通常用它从网络侧获取最新数据，并在获取到最新数据后让 RefreshControl 组件消失。

RefreshControl 支持 View 组件的所有属性，这意味着开发者可以按 View 组件的样式设置它的样式。

- `onRefresh` 是回调函数类型的属性。当 ScrollView 拉到顶部时，这个函数会被执行。
- `refreshing` 是布尔类型的属性，用来设置当前是否应当显示 RefreshControl 组件。

代码 8-1 简单地示范了 RefreshControl 组件的使用。如果要正式使用这个组件，需要将它 `refreshing` 属性与状态机变量挂接。

8.1.5.1 Android 平台特有属性

- `colors` 是数组类型的属性，用来设定刷新指示器的颜色。数组中的每一个元素都应当是一

个颜色值，并且至少需要有一个元素。

- `enabled` 是布尔类型的属性，用来控制是否打开刷新功能。
- `progressBackgroundColor` 是颜色类型的属性，用来设置刷新指示器的背景颜色。
- `size` 是 `RefreshLayoutConsts.SIZE` 类型的属性，用来设置刷新指示器的尺寸，通常不设置它。它的默认值是 `RefreshControl.SIZE.DEFAULT`。

8.1.5.2 iOS 平台特有属性

- `tintColor` 是颜色类型的属性，用来设置刷新指示器的颜色。
- `title` 是字符串类型的属性，用来设置显示在刷新指示器下的字符串。

8.1.6 ScrollView 组件基本用法

在使用 `ScrollView` 组件时，有一点需要注意：`ScrollView` 组件必须要有明确的高度值限制。这个限制要么在 `ScrollView` 组件的样式中设置（不鼓励这种做法），要么在它的父组件（或者更高层组件）样式中设置。如果它的所有父组件的样式中都没有设置高度，将会导致应用出错退出。代码 8-1 示范了 `ScrollView` 组件的使用方法。

代码 8-1, `index.android.js` 或者 `index.ios.js`:

```
'use strict';
var React = require('react-native');
var {
  AppRegistry, StyleSheet, Text, View,
  TouchableHighlight, ScrollView, RefreshControl
} = React;
var Project20 = React.createClass({
  _onScrollBeginDrag:function() {
    console.log('begin drag');
  },
  _onScrollEndDrag:function() {
    console.log('end drag');
  },
  _onMomentumScrollBegin:function() {
    console.log('_onMomentumScrolBegin ');
  },
  _onMomentumScrollEnd:function() {
    console.log('_onMomentumScrolEnd ');
  },
  _onRefresh:function() {
    console.log('_onRefresh is called');
  },
  render: function() {
    return (
      <View style={styles.container}>
        <ScrollView style={styles.scrollView} //声明 ScrollView 组件及其属性
          onMomentumScrollBegin={this._onMomentumScrollBegin}
          onMomentumScrollEnd={this._onMomentumScrollEnd}
          onScrollBeginDrag={this._onScrollBeginDrag}
          onRefresh={this._onRefresh}
          refreshControl={
            <RefreshControl
              title="Pull to Refresh"
              tintColor="#000000"
              progressBackgroundColor="#000000"
              size={RefreshControl.SIZE.DEFAULT}
            />
          />
        </ScrollView>
      </View>
    );
  }
});
AppRegistry.registerComponent('Project20', () => Project20);
```

```

refreshControl={
    <RefreshControl
        refreshing={true}
        onRefresh={this._onRefresh}
        tintColors="#ff0000"
        title="Loading..."
        colors={['#ff0000', '#00ff00', '#0000ff']}
        progressBackgroundColor="#ffff00"/>
    onScrollEndDrag={this._onScrollEndDrag}>
    <View style={styles.aView} />
        <ScrollView horizontal={true} //再声明一个横向的 ScrollView
            style={styles.midScrollView}>
                <View style={styles.bView} />
                <View style={styles.bView} />
            </ScrollView>
        <View style={styles.aView} />
    </ScrollView>
</View>
);
}
});
var styles = StyleSheet.create({
    container: {
        flex: 1,
        backgroundColor: 'gray',
    },
    scrollView: {
        backgroundColor: '#CCCCCC',
    },
    midScrollView: {
        height: 150,
        borderWidth: 1,
        borderColor: 'black',
    },
    aView: {
        margin: 1,
        padding: 0,
        backgroundColor: '#eaeaea',
        height: 375,
    },
    bView: {
        flex: 1,
        height: 148,
        width: 300,
        borderWidth: 1,
        borderColor: 'black',
        backgroundColor: 'grey',
    }
});
AppRegistry.registerComponent('Project20', () => Project20);

```

代码 8-1 在应用的主 View 中放置了一个 ScrollView，并且没有声明它的高度，只是简单地声明了它的背景颜色。在没有声明高度的情况下，ScrollView 将自动占满父组件剩下的高度。在

ScrollView 组件中，先放置了一个普通的 View，然后又放置了一个横向的 ScrollView，并在横向的 ScrollView 中放入了两个子 View。

如图 8-1 所示是代码 8-1 运行效果。



图 8-1 代码 8-1 运行效果

注意：ScrollView 上、下或者左、右拉到头时有动画提示效果。

ScrollView 的用法就介绍到这里，它的作用基本上就是在 ScrollView 的子 View 组件里再放置其他子组件。有兴趣的读者可以修改例程，在 ScrollView 组件的子 View 中加入其他的 React Native 组件。

8.2 ListView 组件

React Native 框架提供了 ListView 组件，让开发者可以高效地展示一个可以垂直滚动的数据列表，并且能高效地更改刷新列表中的数据。

ListView 组件支持列表的一些高级特性，比如给每段/组（section）数据添加一个分段头部，在列表头部和尾部增加单独的内容等。

React Native 认识到 ListView 是手机 UI 设计中非常重要、基础的元素，对其进行了大量的优化工作，使 ListView 能适应动态加载非常多的数据，让它的滚动尽可能平滑。

ListView 组件自身不需要进行样式设置，也不接受被设置样式属性。开发者应当对 ListView

中的每一列设置样式，而不是对 ListView 本身设置样式。

ListView 组件继承了 View 组件与 ScrollView 组件的所有属性。除此之外，它还有自己特殊的属性。

8.2.1 ListView 组件的回调函数

- **onEndReached** 回调函数在 ListView 的所有行都被渲染在屏幕上并且 List 被滑动到底部时将被调用。它不接收任何参数，也无须返回值。底部可以通过 **onEndReachedThreshold** 这个数值型属性定义一个以 pt 为单位的门限值。
- **renderFooter** 回调函数用来渲染 List 的底栏（如果提供了这个回调函数）。它不接收任何参数，需要返回一个 renderable 组件。它在每一次重新渲染时都会被调用。如果渲染底栏的代价太高，则可以将它放在一个静态的容器中置于屏幕底部或者使用其他适合的机制。
- **renderHeader** 回调函数用来渲染 List 的首栏。它不接收任何参数，需要返回一个 renderable 组件。它也一样在每一次重新渲染时都会被调用。也可以用上一段描述的思路优化。
- **renderRow** 回调函数用来渲染 List 的每一栏。它接收 4 个参数：**rowData**、**sectionID**、**rowID**、**highlightRow**，并且需要返回一个 renderable 组件。这 4 个参数是开发者给 ListView 提供的数据、数据的 **sectionID**（由开发者提供）、数据的 **rowID**（数据在列表中第几行）和一个 **highlightRow** 函数。从名字上就可以知道，**highlightRow** 函数是用来高亮显示列表中某行时的调用函数。但目前还没有实现，开发者如果获取这个参数的值，只会获取到一个 **undefined**。
- **renderScrollComponent** 回调函数用来提供一个可滑动的组件供 List 在其中渲染。它接收 **props** 为参数，并且需要返回一个可渲染的组件。默认的 **renderScrollComponent** 返回一个 **ScrollView** 组件。
- **renderSectionHeader** 回调函数用来渲染某一节的头部。它接收 **sectionData** 和 **sectionID** 为参数，并需要返回一个可渲染的组件。如果提供了这个属性，则会向指定节提供一个固定的头部。固定是指在这一节没有结节前，这个头部都会显示在列表顶端，直到这一节的所有项都滑出屏幕。我们将在例程中看到这个效果。
- **renderSeparator** 回调函数如果存在，它将用来在每一行下（但如果最后一行下是另一节的头部将不会渲染）渲染一个分隔单元。它接收 **sectionID**、**rowID**、**adjacentRowHighlighted** 三个参数，并且返回一个 renderable 组件。
- **onChangeVisibleRows** 回调函数用来通知哪些行已经成为屏幕上的可见行，而哪些行的可见性已经改变了。它接收两个参数：**visibleRows**、**changedRows**，不需要返回值。**visibleRows** 是一个 map 结构，它的格式是 { **sectionID**: { **rowID**: true } } 记录了当前屏幕上所有可见行的 ID。**changedRows** 是一个 map 结构，它的格式是 { **sectionID**: { **rowID**: true | false } }，记录了所有被改变的能否可见行的 ID，**rowID** 为 true 时，表示此行现在可见（进入了屏幕）；false 表示此行现在不可见（滑出了屏幕）。

8.2.2 ListView 组件的其他属性

- `dataSource` 是一个 `ListViewDataSource` 类型的属性，用来描述列表的数据来源。我们将在接下来的示例中看到如何定义数据来源。
- `initialListSize` 是数值类型的属性定义了多少行会在 `ListView` 挂接上时被渲染。使用这个属性用来保证第一屏的 `List` 会一次性被渲染，而不是通过多次的帧刷新显示出来。
- `onEndReachedThreshold` 是数值类型的属性，需要结合 `onEndReached` 回调函数使用。它以 `pt` 为单位。
- `pageSize` 是数值类型的属性，定义了在一个事件循环中，多少行会被渲染。在默认情况下，它的值是 1，即在每次消息循环中只有一行会被渲染。这种把可能占用 CPU 处理时间较长的工作分散成多个耗时少的工作的做法，可以让程序运行得更加流畅、用户体验更好。
- `removeClippedSubviews` 是布尔类型的属性，无默认值。当它为 `true` 时，React Native 框架通过使不在屏幕范围内的行暂不处理计算而提高划动效果体验。它需要与每行容器中的 `overflow:hidden` 配合使用。这个函数属性还在实验阶段，如果开发者发现使用有问题，请不要使用它。
- `scrollRenderAheadDistance` 是数值类型的属性，定义了当一个列表行距出现在屏幕最下方（或者最上方）还有多少 `px` 时，`ListView` 组件就应当开始渲染该列表行。普通开发者无须关心此值。
- `automaticallyAdjustContentInsets` 是布尔类型的属性，它目前还没有出现在官方文档上。如果开发者发现自己的 `ListView` 与其上一个组件之间有奇怪的空格，则尝试将这个属性设为 `false` 以纠正这个问题。

8.2.3 ListView 组件的成员函数

在某些情况下，开发者可能需要调用 `ListView` 组件的 `scrollTo` 函数，用于将当前列表滚动到指定位置。这个函数的原型是：

```
scrollTo: function(destY, destX)
```

只要拿到 `ListView` 组件的引用，就可以直接调用这个函数。调用函数时传入的两个参数是指定位置的坐标值（`X` 值传入 0）。

8.3 简单的列表

简单的列表是相对于 8.4 节中带有分段标志的列表而言的。简单的列表效果图见图 8-1。在图 8-1 中，图标周围有一圈白色，不是很美观，但是美工可以通过“背景色”手段轻松地解决这个问题。受限于本书黑白印刷，界面还是只有黑、白、灰三种颜色。

读者学习到本章时，已经可以利用前面学习到的知识，通过 React Native 框架开发出很复杂、精致的移动应用 UI 了。但是有一个前提，那就是：如果要开发复杂、精致的移动应用 UI，则需要一个美工，一个经过正规美术、设计相关专业系统学习的美工能极大地美化移动应用界面。美

工会提供给你每个界面的设计图，细致到每个区域的每张图片、背景色、透明度、距离、边框空隙与填充设计、圆角率等参数。



图 8-2 简单的列表显示效果

利用 ListView 在手机界面上实现列表，开发者需要做如下工作。

8.3.1 准备列表的数据源

列表的数据源通常是某个类的数组。数组有多少个元素，列表就会有多少行。数组中每一个对象的成员变量中的数据，都可以展现在列表的行中。

8.3.2 声明状态机变量

声明渲染 ListView 用的 DataSource 类型变量为某组件的状态机变量。

通常声明语句如下：

```

.....//在某组件的 getInitialState 函数中
状态机变量名:new ListView.DataSource({
    rowHasChanged: (oldRow, newRow) => oldRow !== newRow,
}),
.....//继续声明其他状态机变量

```

在初始化状态机变量时，我们新建了一个 ListView.DataSource 类型的对象。在建立这个对象时，需要为它提供 rowHasChanged 成员函数的实现。React Native 框架将使用这个成员函数来判断列表的某一行是否需要重新渲染。我们通过 `(oldRow, newRow) => oldRow !== newRow` 这个函数中的严格不等于来判断老的行数据与新的行数据是否不同，如果不同，则返回 `true`，表示需要重新渲染该行；否则不需要重新渲染该行。开发者通常不需要更改这个成员函数的实现，除非要满足某些特殊需求。

通过 DataSource 对象的 rowHasChanged 成员函数，ListView 组件实现了只更新数据有改变的

行,从而达到了高效渲染列表的目的,让列表刷新、滚动更平滑,对 CPU 时间、内存的占用更小。

8.3.3 将数据源中的数据拷贝到 DataSource 中

ListView 要求开发者将列表中需要显示的数据从数据源中拷贝到 DataSource 中。假设组件中 DataSource 类型的状态机变量为 listDS,实现拷贝的语句如下:

```
.....
this.setState(()=>{
  return {
    listDS: this.state.listDS.cloneWithRows(数据源变量名)
  };
});
.....
```

因为 JavaScript 的数据传递机制是“按共享传递”,这种拷贝不会大幅增加内存开销,只是相当于多声明了一些变量。

大家可以看到,ListView 组件不是直接使用开发者准备好的数据源的,而是使用 cloneWithRows 函数对数据源中的所有数据进行复制。当数据源中的数据被改变时,JavaScript 的“按共享传递”机制保证了 ListView 组件的 DataSource 中的数据不会被改变,直到开发者再次通过 cloneWithRows 函数同步 DataSource 中的数据与数据源中的数据。

因为数据源不是直接参与 React Native 框架的界面渲染过程的,因此数据源不应当放在任何组件的状态机变量中。

8.3.4 定义如何渲染列表中的每一行

这一步就像我们在日记例程(上)假列表中所做的那样,写出如何展示一行中的各个数据。需要将渲染的步骤写成一个函数,这个函数返回一个可渲染的 JSX 结构。

在函数的三个参数中,log 是一个对象,数据源中数组的某一个元素,开发者用它来渲染列表中需要填入的数据。sectionID 是分段标识,对于简单的列表,开发者可以不去管这个参数。rowID 表示当前渲染的行在列表中位于第几行(从 0 开始,与数据源数组下标对应)。

```
renderListItem: function(log: {}, sectionID: number, rowID: number) {
  return (
    <TouchableOpacity onPress={()=>this.props.selectLististItem(rowID)}>
      <View style={MCV.secondRow}>
        <Image style={MCV.moodStyle}
          source={log.mood}/>
        <View style={MCV.subViewInReader}>
          <Text style={MCV.textInReader}>
            {log.title}
          </Text>
          <Text style={MCV.textInReader}>
            {this.getTimeString(log.time)}
          </Text>
        </View>
      </View>
    </View>
  );
}
```

```

    </TouchableOpacity>
  );
},

```

8.3.5 实现简单的列表

最后在需要呈现 ListView 的组件中加入 ListView 结构，这个结构需要有两个属性。

```

<ListView
  dataSource={准备好的 DataSource 状态机变量或者传递下来的属性}
  renderRow={准备好的渲染每一行的函数}
/>

```

代码 8-2 在 DiaryList 组件中使用了 ListView 基础组件。

代码 8-2, DiaryList.js:

```

'use strict';
var React = require('react-native');
let MCV = require('./MCV');
var {
  View, Text, TextInput, TouchableOpacity, Image, ListView
} = React;
let DiaryList = React.createClass({
  keyword:'',
  getInitialState: function() {
    return {
      buttonText: '点击搜索日记列表'
    }
  },
  render: function() {
    return (
      <View style={MCV.container}>>
        <View style={MCV.firstRow} > //下面的代码使用到了三元运算符，参见附录 A.8
          { (Platform.OS === 'android' )? //视平台不同显示不同的组件
            ( //Android 平台显示组件
              <View style={{borderWidth:1}}>
                <TextInput autoCapitalize="none"
                  placeholder='输入搜索关键字'
                  clearButtonMode="while-editing"
                  onChangeText={this.updateSearchKeyword}
                  style={MCV.searchBarTextInput}/>
              </View>
            ):( //iOS 平台显示组件
              <TextInput autoCapitalize="none"
                placeholder=' 输入搜索关键字'
                clearButtonMode="while-editing"
                onChangeText={this.updateSearchKeyword}
                style={MCV.searchBarTextInput}/>
            )
          } //视平台不同显示不同的组件结束
        <TouchableOpacity onPress={this.props.writeDiary} >
          <Text style={MCV.middleButton}>
            写日记
          </Text>
        </TouchableOpacity>
      </View>
    );
  }
});

```

```

        </Text>
      </TouchableOpacity>
    </View>
    //不同于日记例程（上）中这里使用一个假列表，现在这里使用 ListView 组件
    <ListView dataSource={this.props.diaryListDataSource}
      //dataSource 描述列表的数据，在本例中，它来源于上层组件
      renderRow={this.renderListItem}>
    //renderRow 描述如何渲染列表中的每一行，在本例中，它挂接 renderListItem 函数
    </ListView>
  </View>
);
},
// renderListItem 函数定义了如何渲染列表中的每一行，它的三个参数由 React Native 框架提供
renderListItem: function(log: {}, sectionID: number, rowID: number) {
  //log 是一个对象，代表当前行的相应数据，由开发者通过 dataSource 提供
  //sectionID 代表当前列表的分段号，8.4 节中会使用这个分段号
  //rowID 代表当前行在整个列表中的行号
  return (
    <TouchableOpacity onPress={()=>this.props.selectLististItem(rowID)}>
      //使用 TouchableOpacity 将列表中的每一行声明为可按的控件
      //并且指定按下事件的处理函数，按下事件上报时会带上本行在列表中的行号
      <View style={MCV.secondRow}>
        //为了显示美观，请读者在 MCV.js 的 secondRow 样式定义中加入：
        //backgroundColor: 'grey', borderRadius:4, margin:1
        <Image style={MCV.moodStyle}
          source={log.mood}/> //心情图片被传入
        <View style={MCV.subViewInReader}>
          <Text style={MCV.textInReader}>
            {log.title} //日记标题被传入
          </Text>
          <Text style={MCV.textInReader}>
            {this.getTimeString(log.time)} //将日记时间处理后传入
          </Text>
        </View>
      </View>
    </TouchableOpacity>
  );
},
//getTimeString 函数接收一个代表时间的字符串，处理后传回一个代表时间的字符串用于列表显示
getTimeString:function( aString ) {
  let ctime = new Date(aString);
  return ''+ctime.getFullYear()+ '年'+(ctime.getMonth()+1)+'月'+ctime.getDate()
    +'日 星期'+(ctime.getDay()+1)+' '+ctime.getHours()+ ':' +ctime.getMinutes();
}
});
module.exports=DiaryList;

```

代码 8-3 是配套的 index.android.js 或者 index.ios.js。

代码 8-3:

```

'use strict';
let React = require('react-native');
let {
  AppRegistry, StatusBarIOS, AsyncStorage, ListView

```

```

} = React;
let DiaryList=require('./DiaryList');
let DiaryWriter=require('./DiaryWriter');
let DiaryReader=require('./DiaryReader');
let angryMood = require('./image/angry.jpg')
let peaceMood=require('./image/peace.jpg');
let happyMood=require('./image/happy.jpg');
let sadMood=require('./image/sad.jpg');
let miseryMood=require('./image/misery.jpg');
let Project19 = React.createClass({
  realDairyList:null, //日记例程（下）中实现了真正的日记列表，但我们还是在
                        // Project19 的成员变量中保存一份日记列表，在状态机中保存另一份

  listIndex:0,
  bubbleSortDiaryList:function() {
    .....// bubbleSortDiaryList 函数与日记例程（上）中的实现一样，在此不再列出
  },
  getInitialState: function() {
    AsyncStorage.getAllKeys().then(
      (keys)=>{
        AsyncStorage.multiGet(keys).then(
          (results)=>{
            let resultsLength=results.length;
            this.realDairyList = Array();
            for(let counter = 0;counter<resultsLength;counter++) {
              this.realDairyList[counter]=JSON.parse(results[counter][1]);
            }
            this.bubbleSortDiaryList();
            if (resultsLength>0) {
              resultsLength--;
              this.listIndex=resultsLength;
              let newMoodIcon;
              switch(this.realDairyList[resultsLength].mood) {
                case 2:
                  newMoodIcon=angryMood;
                  break;
                case 3:
                  newMoodIcon=sadMood;
                  break;
                case 4:
                  newMoodIcon=happyMood;
                  break;
                case 5:
                  newMoodIcon=miseryMood;
                  break;
                default:
                  newMoodIcon=peaceMood;
              }
              let newtitle = this.realDairyList[resultsLength].title;
              let newbody = this.realDairyList[resultsLength].body;
              let ctime = new Date(this.realDairyList[resultsLength].time);
              let timeString = ''+ctime.getFullYear()+''年'+(ctime.getMonth()+1)+'
月'+ctime.getDate()+''日 星期'+(ctime.getDay()+1)+' '+ctime.getHours()+':'+ctime.getMinutes();
              this.setState(()=>{
                return {
                  diaryListDataSource:

```

```

this.state.diaryListDataSource.cloneWithRows(this.realDairyList),
//在这里将成员变量中的日记列表同步到状态机变量中的日记列表
        diaryMood: newMoodIcon,
        diaryTime: timeString,
        diaryTitle:newtitle,
        diaryBody: newbody
    };
    });
}
else {
    this.setState(()=>{
        return {
            diaryTime: '没有历史日记',
            diaryTitle:'没有历史日记',
            diaryBody: ''
        }
    });
}
},
(errors)=>{
    console.log( '1st error:'+errors[0].message);
}
).catch((error)=>{
    console.log( ' then error:'+error);
}
);
},
(error)=>{
    console.log( ' getAllKeys error:'+error.message);
}
);
console.log( 'getInitialState return reading. ');
return {
    uiCode: 1,
    diaryListDataSource:new ListView.DataSource({
        rowHasChanged: (oldRow, newRow) => oldRow !== newRow
    }),
//声明状态机列表中的 ListView 数据来源 diaryListDataSource。它需要一个 JSON 数据
//指示列表是否已经被改变。rowHasChanged 键对应的值要求是一个函数
//我们通过 Project19 组件的 diaryListChanged 函数来指示列表数据来源是否已经被改变
    diaryMood: peaceMood,
    diaryTime: '读取中.....',
    diaryTitle:'读取中.....',
    diaryBody: '读取中.....'
};
},
readingPreviousPressed: function() {
    .....// readingPreviousPressed 函数与日记例程（上）中的实现一样，在此不再列出
},
readingNextPressed: function() {
    .....// readingNextPressed 函数与日记例程（上）中的实现一样，在此不再列出
},

```



```

returnPressed: function() {
    .....// returnPressed 函数与日记例程（上）中的实现一样，在此不再列出
},
saveDiaryAndReturn: function(newDiaryMood,newDiaryBody,newDiaryTitile) {
    let cTime = new Date();
    let timeString = '' + cTime.getFullYear() + '年' + (cTime.getMonth() + 1) +
        '月' + cTime.getDate() + '日 星期' + (cTime.getDay()+1)+' '
        + cTime.getHours() + ':' + cTime.getMinutes();
    let newMoodIcon;
    switch(newDiaryMood) {
        case 2:
            newMoodIcon=angryMood;
            break;
        case 3:
            newMoodIcon=sadMood;
            break;
        case 4:
            newMoodIcon=happyMood;
            break;
        case 5:
            newMoodIcon=miseryMood;
            break;
        default:
            newMoodIcon=peaceMood;
    }
    let aDiary = Object();
    aDiary.title = newDiaryTitile;
    aDiary.body = newDiaryBody;
    aDiary.mood = newMoodIcon;
    aDiary.time = cTime;
    aDiary.sectionID = '' + cTime.getFullYear()+''+cTime.getMonth()+1+'月';
    aDiary.index= Date.parse(cTime);
    AsyncStorage.setItem( ''+aDiary.index, JSON.stringify(aDiary)).then(
        ()=>{
            console.log( 'Saving succeed');
        },
        (error)=>{
            console.log( 'Saving failed, error' + error.message);
        }
    );
    let totalLength=this.realDairyList.length;
    this.realDairyList[totalLength]=aDiary;
    this.listIndex = totalLength;
    this.setState(()=>{
        return {
            uiCode:1,
            //状态机变量中的日记列表数据来源同步日记列表数据
            diaryListDataSource:
this.state.diaryListDataSource.cloneWithRows(this.realDairyList),
            diaryTime: timeString,
            diaryTitle: newDiaryTitile,
            diaryMood: newMoodIcon,
            diaryBody: newDiaryBody
        };
    });
});

```

```

    },
    writeDiary: function() {
      .....// writeDiary 函数与日记例程（上）中的实现一样，在此不再列出
    },
    searchKeyword:function(keyword) {
      console.log( 'new search keyword is:' + keyword );
    },
    selectLististItem: function(aIndex) {
      //现在 selectListItem 可以知道用户按下了日记列表中的第 aIndex 行
      this.listIndex = aIndex;          //记录下用户按下了第几行（或者说选择了第几篇日记）
      let newDiaryTitile=this.realDairyList[aIndex].title;
      let newDiaryBody=this.realDairyList[aIndex].body;
      let newMoodIcon;
      switch(this.realDairyList[aIndex].mood) {
        case 2:
          newMoodIcon=angryMood;
          break;
        case 3:
          newMoodIcon=sadMood;
          break;
        case 4:
          newMoodIcon=happyMood;
          break;
        case 5:
          newMoodIcon=miseryMood;
          break;
        default:
          newMoodIcon=peaceMood;
      }
      let currentTime = new Date(this.realDairyList[aIndex].time);
      let timeString = '' + currentTime.getFullYear() + '年' + (currentTime.getMonth()
+ 1) + '月' + currentTime.getDate() + '日 星期' + (currentTime.getDay()+1)+'
'+currentTime.getHours()+ ':' + currentTime.getMinutes();
      this.setState(()=>{
        return{
          uiCode:2,
          diaryTime: timeString,
          diaryTitle: newDiaryTitile,
          diaryMood: newMoodIcon,
          diaryBody: newDiaryBody
        };
      });
    },
    showDiaryList:function() {
      return (
        <DiaryList fakeListTitle={this.state.diaryTitle}
          selectLististItem={this.selectLististItem}
          searchKeyword={this.searchKeyword}
          writeDiary={this.writeDiary}
          diaryListDataSource={this.state.diaryListDataSource}/>
        //将真正的日记列表数据源传入，而不是原来假的数据
      );
    },
    showDiaryWriter:function() {
      .....//showDiaryWriter 函数与日记例程（上）中的实现一样，在此不再列出
    }
  },
  showDiaryWriter:function() {
    .....//showDiaryWriter 函数与日记例程（上）中的实现一样，在此不再列出
  }
}

```

```

    },
    showDiaryReader:function() {
        .....//showDiaryReader 函数与日记例程（上）中的实现一样，在此不再列出
    },
    componentWillMount:function() {
        if ( StatusBarIOS != null ) StatusBarIOS.setHidden(true);
    },
    render: function() {
        .....// render 函数与日记例程（上）中的实现一样，在此不再列出
    },
  });
  AppRegistry.registerComponent('Project19', () => Project19);

```

运行代码，日记列表界面显示效果如图 8-2 所示。

8.3.6 列表栏的高级处理

通过在 ListView 的 `renderRow` 指向的渲染函数中，用可触摸组件包含每一列，使得每一列都成为一个整体可触摸的组件。代码 8-2 演示了这一点。当点击一个列表栏时，可以通过 `rowID` 得到这个列表栏的行号与在列表中的位置。

在更灵活的用法中，可以在每一列中定义多个可触摸组件，让每一列可以触发多个触摸事件。

如果在列表栏中放置一个可选取组件，则可以让列表成为一个可选取的列表。

目前官方还没有实现 `renderRow` 函数中的高亮列表栏特性，但列表栏是由开发者负责定义如何渲染的。通过对 `renderRow` 函数传入的数据进行判断，判断出当前列表栏所渲染的数据是需要特殊显示的数据，就可以返回特殊的 JSX 代码，让这一行与其他行显示不同，比如可以高亮显示，以及比其他行的高度更高等。

8.4 带分段标志的列表

React Native 的 ListView 组件可以给列表设置分栏和尾栏。在讨论如何实现前，让我们先看一看什么是分栏和尾栏。

开发者可以把列表中有明显相同特征的多行放入一个分栏中，在这些列表行之上显示一个分栏区域，比如在电话记录列表中，对不同天的记录可以按天分栏；又比如在日记例程中，可以把同一个月记的日记分为一栏，按月分栏。

ListView 组件在 iOS 和 Android 两个平台上绝大部分的表现都是一样的，只是在分栏显示上，iOS 和 Android 平台上的呈现有些差异。在这里描述一下。当列表刚开始展示时，第一个分栏区域（由开发者指定的代码绘制）会出现在列表的最顶端。在 iOS 手机上，当用户向上划动列表时，第一个分栏区域会保持在列表的最顶端不动，而是下方的列表动，直到第一个分栏中所有的行都上移出屏幕，第一个分栏区域此时会被第二个分栏区域顶出列表顶端，第二个分栏会保持在列表的最顶端区域不动……图 8-3 显示了在 iOS 平台上有分栏的列表效果。

在 Android 手机上，当列表刚开始展示时，第一个分栏区域（由开发者指定的代码绘制）会

出现在列表的最顶端。当用户上下划动列表时，分栏会跟随列表一起上下移动（而不是像在 iOS 手机上那样保持在列表的顶端不动）。图 8-3 显示了在 Android 平台上有分栏的列表效果。



图 8-3 在 iOS 平台上分栏固定的效果



图 8-4 在 Android 平台上分栏随列表上下移动的效果

尾栏是由开发者指定代码渲染的一块区域。它作为列表的尾部，显示在列表的最下方。iOS 和 Android 手机在尾栏显示上没有区别。图 8-4 显示了尾栏在列表中的显示效果。

在 8.3 节的基础上,给列表加上分栏需要如下步骤。

8.4.1 准备数据源

在 8.3.1 节中我们只准备了一个数据源,这可以视为该数据源中的所有数据都属于同一个分栏。而现在的列表有多个分栏,这就意味着有多少个分栏,就要准备多少个数据源。



图 8-5 ListView 组件尾栏效果

8.4.2 声明状态机变量

声明渲染 ListView 用的 DataSource 类型变量为某组件的状态机变量。

通常声明语句如下:

```
.....//在某组件的 getInitialState 函数中
状态机变量名:new ListView.DataSource({
  rowHasChanged: (oldRow, newRow) => oldRow !== newRow,
  sectionHeaderHasChanged:(oldSH, newSH)=>oldSH!==newSH
})
```

```

  }),
  .....//继续声明其他状态机变量

```

在初始化状态机变量时，我们新建了一个 `ListView.DataSource` 类型的对象。在建立这个对象时，除了在 8.3.2 节中讨论的需要为它提供 `rowHasChanged` 成员函数的实现外，还需要提供 `sectionHeaderHasChanged` 成员函数的实现。React Native 框架将使用这个成员函数来判断列表的某分栏是否需要重新渲染。我们通过 `(oldSH, newSH)=>oldSH!==newSH` 这个函数中的严格不等于来判断老的分栏数据与新的分栏数据是否不同，如果不同，则返回 `true`，表示需要重新渲染该分栏；否则不需要重新渲染。开发者通常不需要更改这个成员函数的实现，除非是为了满足移动应用的某些特殊需求。

8.4.3 将数据源中的数据拷贝到 DataSource 中

ListView 要求开发者将列表中需要显示的数据从数据源拷贝到 `DataSource` 中。假设组件中 `DataSource` 类型的状态机变量为 `listDS`，实现拷贝的语句如下：

```

.....
let newListWithSection = []; //JavaScript 动态增加类的成员变量的技巧
let sec1 = '分区 1';          //在本例程中是写死了增加三个，在实际开发中，
let sec2 = '分区 2';          //可以视需要动态任意增加
let sec3 = '分区 3';
newListWithSection[sec1]=this.realDairyList;
newListWithSection[sec2]=this.realDairyList;
newListWithSection[sec3]=this.realDairyList;
let fakeSections=[sec1, sec2, sec3]; //数组在本例程中是写死的，但 JavaScript 也支持动态增加
.....
this.setState(()=>{
  return {
    listDS: this.state.listDS.cloneWithRowsAndSections(newListWithSection,fakeSections),
  };
});
.....

```

在这段代码中，使用到了 JavaScript 动态声明类的成员变量的技巧。

与 8.3.3 节相比，ListView 组件不是直接使用开发者准备好的数据源，而是使用 `cloneWithRowsAndSections` 函数对数据源中的所有数据进行复制，除了提供数据源外，还需要提供分段数据。当数据源中的数据被改变时，JavaScript 的“按共享传递”机制保证了 ListView 组件的 `DataSource` 中的数据不会被改变，直到开发者再次通过 `cloneWithRowsAndSections` 函数同步 `DataSource` 中的数据和数据源中的数据。

8.4.4 定义如何渲染每个分栏

既然实现的是带分栏的列表，那么渲染分栏是必不可少的。开发者通过 `renderSectionHeader` 回调函数定义如何渲染每个分栏，这个函数返回一个可渲染的 JSX 结构。

在函数的两个参数中，`sectionData` 是一个对象，对应着本分栏的所有数据；`sectionID` 是一个字符串，代表着分栏字符串。

```
renderSectionHeader: function(sectionData, sectionID) {
  return (
    <View style={MCV.section}>
      <Text style={MCV.sectionText}>{sectionID}</Text>
    </View>
  )
},
```

8.4.5 定义如何渲染首、尾栏

开发者可以给列表加上一个首栏和一个尾栏，首栏会固定显示在列表首，而尾栏会固定显示在列表尾。它们都没有任何参数，需要返回一个可渲染的 JSX 结构。

不管是简单列表还是复杂列表，都可以有首、尾栏。

```
renderHeader: function() {
  return (
    <View style={MCV.section}>
      <Text style={MCV.sectionText}>我是 Header</Text>
    </View>
  )
},
renderFooter: function() {
  return (
    <View style={MCV.section}>
      <Text style={MCV.sectionText}>我是 Footer</Text>
    </View>
  )
},
```

8.4.6 列表间隔渲染

开发者可以给 `ListView` 组件提供 `renderSeparator` 回调函数，在这个回调函数中定义如何渲染列表中每个元素的间隔。这个回调函数没有参数。

```
renderSeparator: function() {
  return (
    <View style={MCV.section}>
      <Text style={MCV.sectionText}>我是 Separator</Text>
    </View>
  )
},
```

8.4.7 实现带分段标志的列表

最后需要呈现 `ListView` 的组件中加入 `ListView` 结构，这个结构至少需要有三个属性。

```
<ListView
  dataSource={this.props.diaryListDataSource}           //必须提供
  renderRow={this.renderListItem}                       //必须提供
```

```
renderSectionHeader={this.renderSectionHeader}           //必须提供
renderSeparator = {this.renderSeparator }
renderHeader={this.renderHeader}
renderFooter={this.renderFooter}/>
```

8.4.8 总结

在实现 ListView 时，开发者可以按照应用需要，灵活定义如何渲染每一个 ListView 的部件。通常列表中的元素、间隔、分栏都是等高的，有着相同的格式，但不是必须如此。开发者完全可以让每一个元素按照需要在显示上有不同的高度、不同的格式及内容。通常列表中的列表项都是左侧有一张图片，右侧是相关信息，但开发者完全可以让一个列表项显示多张图片，而不显示任何文字。

8.5 日记例程（下）总结

这个日记例程还有许多需要完善的功能。比如搜索功能就完全没有实现，列表分段应当按照写日记的年份和月份来做，但是因为无法很快看到效果，本书也只是说明怎么实现这个功能，然后用几个假分段去实现了。这些功能请读者利用本书前面章节中讨论的技术自行实现。

第 9 章

等待提示条、进度条和 Switch

等待提示条、进度条和开关是常见的手机 UI 控件，并且在它们的发展过程中，iOS 平台和 Android 平台发展出来了两套显示风格不同的控件。React Native 框架尊重用户的使用习惯，为这些控件提供了分平台的基础组件，开发者通过平台自适应代码，就可以开发出符合手机用户使用习惯的移动应用界面。

9.1 ProgressBarAndroid 组件

React Native 框架提供了 ProgressBarAndroid 组件供开发者在 Android 手机上实现等待提示条和进度条。

9.1.1 ProgressBarAndroid 组件样式设置

ProgressBarAndroid 组件支持所有的 View 组件的样式设置，开发者可以将其视为一个 box，使用 flexbox 的各种样式设置和 View 的样式设置。

9.1.2 ProgressBarAndroid 其他属性

ProgressBarAndroid 组件还支持 View 组件的其他非样式设置属性。除此之外，它的属性还有：

- color，用来指定等待提示条或者进度条的颜色。
- indeterminate，布尔类型的属性，用来指定当前是等待提示条还是有明确进度的进度条。默认值是 true，表示当前的组件是等待提示条。此参数只能与 styleAttr='Horizontal'配合使用。
- progress，数值类型的属性，用来指定进度条的进度，它的取值必须在 0 和 1 之间。
- styleAttr，字符串类型的属性，用来指定 ProgressBar 的外形。它可以取值为：Horizontal、Small、Large、Inverse、SmallInverse、LargeInverse。

9.1.3 Android 平台等待提示条

最简单的 Android 平台等待提示条组件例程见代码 9-1。注意 ProgressBarAndroid 组件需要通过 require 语句来获得。

代码 9-1:

```
'use strict';
var ProgressBar = require('ProgressBarAndroid');
var React = require('react-native');
var { AppRegistry } = React;
var MovingBar = React.createClass({
  render: function() {
    return <ProgressBar/>;
  },
});
AppRegistry.registerComponent('Project19', () => MovingBar);
```

代码 9-1 运行效果如图 9-1 所示（手机界面顶部截图，下面都是空白）。在界面的最上方显示一个不断转动的等待提示条。

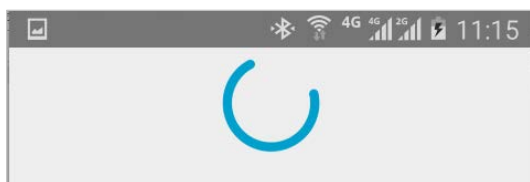


图 9-1 Android 手机等待提示组件运行效果（一）

开发者还可以通过让 `ProgressBar` 成为某个 `View` 的子组件方式，让等待提示条显示在界面的其他地方。

使用 `<ProgressBar styleAttr="Small"/>` 的显示效果如图 9-2 所示。



图 9-2 Android 手机等待提示组件运行效果（二）

使用 `<ProgressBar styleAttr="Large"/>` 的显示效果与图 9-1 相同。

`ProgressBarAndroid` 组件不支持 `Inverse`、`SmallInverse`、`LargeInverse` 这几个 `styleAttr` 属性的取值。也就是说，在 Android 平台上无法实现等待提示条逆时针旋转的动画效果。

Android 平台还支持水平方式的等待提示条。在代码 9-1 中，对 `ProgressBar` 组件加入 `styleAttr="Horizontal"` 后，就可以让等待提示条变成水平方式的。其显示效果如图 9-3 所示。



图 9-3 Android 手机水平进度条效果

在图 9-3 中，进度条动画会一直来回滚动，直到这个组件被 React Native 框架卸载完毕。

9.1.4 React Native 框架中定时器的使用

9.1.4.1 使用 JavaScript 提供的定时器方法

在 React Native 应用中，可以像在 JavaScript 中那样使用定时器（不推荐，建议使用 9.1.4.2 节中的方法）。启动、停止定时器的代码见代码 9-2。

代码 9-2:

```
.....
progressTimer:null,           //声明一个成员变量，准备用它指向定时器
.....
componentDidMount: function() {           //启动定时器，本例中是在组件挂接时启动的，开
                                           //发者按照需要可以在任何时候启动
this.progressTimer=window.setInterval( this.changeProgress, 1000 );
    //第一个参数是定时器到时后调用的函数名称，第二个参数是定时值，以毫秒为单位
    //定时器启动后会每隔指定的时间调用指定的函数一次，直到定时器被关闭
},
componentWillUnmount:function() {
    window.clearInterval(this.progressTimer);    //在需要时关闭定时器
},
```

其他的常用定时器方法这里不再讨论。

如果开发者坚持要使用 JavaScript 的全局定时器，那么一定要记住：如果一个组件开启了一个定时器，在定时器还没超时这个组件被卸载而没有关闭定时器，那么 React Native 应用将在定时器超时时崩溃！

9.1.4.2 使用 react-timer-mixin

使用 JavaScript 提供的定时器方法，要求开发者在使用完定时器后要关闭定时器，否则会面临不可知的错误。React Native 框架为开发者准备了 react-timer-mixin 来避免这个错误。react-timer-mixin 是跨平台的模块，Android 和 iOS 都可以使用。

使用 react-timer-mixin 和普通的 JavaScript 使用定时器的方法很类似。步骤如下：

- 声明需要使用 react-timer-mixin;
- 在组件代码中包含 TimerMixin;
- 使用各定时器方法。

在原来的 JavaScript 代码中调用 setTimeout、setInterval、setImmediate 和 requestAnimationFrame 函数的地方，在函数前加“this.”以调用包含 TimerMixin 后本组件得到的静态方法。react-timer-mixin 的用法参见代码 9-3 和代码 9-6。

代码 9-3:

```
var React = require('react');
var TimerMixin = require('react-timer-mixin'); //声明需要使用 react-timer-mixin
var Component = React.createClass({
    mixins: [TimerMixin],           //在组件代码中包含 TimerMixin
```

```

componentDidMount() {
  this.setTimeout(
    () =>{ console.log('react-timer-mixin 很方便安全!')},    //定义超时时执行函数
    500                //0.5 秒后超时
  );
}
});

```

`requestAnimationFrame` 是一个比较新的 JavaScript 定时器用法,大家可以把它看成一个每秒执行 60 次的定时器。相比 `setTimeout`、`setInterval` 它有许多优点,感兴趣的读者请自行搜索查阅相关资料。

9.1.5 Android 平台进度条

Android 平台进度条的实现方式见代码 9-4。

代码 9-4:

```

'use strict';
var ProgressBar = require('ProgressBarAndroid');
var React = require('react-native');
var {
  AppRegistry,
} = React;
var MovingBar = React.createClass({
  progressTimer:null,    //声明一个成员变量,准备用它指向定时器
  getInitialState: function() {
    return {
      progress: 0        //progress 用来指示进度完成了多少。它的取值为 0~1
    };
  },
  changeProgress:function() {
    let newProgress = (this.state.progress + 0.03) % 1;    //这是模拟进度完成的过程
    this.setState({progress: newProgress});
  },
  componentDidMount: function() {    //启动定时器
    this.progressTimer=window.setInterval( this.changeProgress, 1000 );
  },
  componentWillUnmount:function() {    //停止定时器
    window.clearInterval(this.progressTimer);
  },
  render: function() {
    return <ProgressBar styleAttr="Horizontal"
      progress={this.state.progress}    //指明进度记载变量
      indeterminate={false}/>;
  },
});
AppRegistry.registerComponent('Project19', () => MovingBar);

```

9.2 iOS 进度条组件

9.2.1 ProgressViewIOS 组件样式设置

ProgressViewIOS 组件支持所有的 View 组件的样式设置，开发者可以将其视为一个 box，使用 flexbox 的各种样式设置与 View 的样式设置。但在开发者设置的样式中高度值会自动被忽略。

9.2.2 ProgressViewIOS 其他属性

ProgressViewIOS 组件还支持 View 组件的其他非样式设置属性。除此之外，它的属性还有：

- **Progress**，数值类型的属性，用来指定进度条的进度，它的取值必须在 0 和 1 之间。与 Android 水平进度条不同，这个值必须要设置，并且要有某种机制让它变化，否则界面上就是一个不变的进度条。
- **progressImage**，Image.propTypes.source 类型的属性，用来指定一张可以拉伸的图片。当提供了这个属性后，这张图片将被用来取代默认的进度条进度线。
- **trackImage**，Image.propTypes.source 类型的属性，用来指定一张可以拉伸的图片。当提供了这个属性后，这张图片将被用来显示为进度条的背景图片。
- **progressTintColor**，字符串类型的属性，用来指定进度条的颜色。
- **trackTintColor**，字符串类型的属性，用来指定进度条背景图片可以渲染的侵蚀颜色。
- **progressViewStyle**，字符串类型的属性，这是 React Native 为了将来扩展而预留的属性。截至 React Native 0.18.0 版本，它的取值只可以是 default 或者 bar，并且设置为这两个值之一对组件的显示没有任何区别。

9.2.3 iOS 平台进度条

最简单的 iOS 平台进度条例程见代码 9-5。与 Android 平台的进度条不同，ProgressViewIOS 组件不需要通过 require 语句来获得。

代码 9-5，index.ios.js:

```
'use strict';
var React = require('react-native');
var { AppRegistry, ProgressViewIOS } = React;
var TimerMixin = require('react-timer-mixin'); //声明需要使用 react-timer-mixin
var MovingBar = React.createClass({
  mixins: [TimerMixin], //在组件代码中包含 TimerMixin
  getInitialState() {
    return {
      progress: 0,
    };
  },
  componentDidMount() {
    this.updateProgress();
  },
  updateProgress() {
    var progress = (this.state.progress + 0.0025) % 1;
```

```

    this.setState({ progress });
    this.requestAnimationFrame(() => this.updateProgress()); //每 1/60 秒执行一次
  },
  render() {
    return (
      <ProgressViewIOS progress={this.state.progress}/>
    );
  },
});
AppRegistry.registerComponent('Project19', () => MovingBar);

```

代码 9-5 运行效果与图 9-3 非常相似，这里不再给出。

9.3 iOS 平台等待提示条

9.3.1 ActivityIndicatorIOS 组件样式设置

ActivityIndicatorIOS 组件支持所有的 View 组件的样式设置，开发者可以将其视为一个 box，使用 flexbox 的各种样式设置与 View 的样式设置。

9.3.2 ActivityIndicatorIOS 其他属性

ActivityIndicatorIOS 组件还支持 View 组件的其他非样式设置属性。除此之外，它的属性还有：

- **animating**，布尔类型的属性，用来指示等待提示条是否旋转。当它为 **True**（默认值）时，等待提示条将保持不停地旋转。
- **color**，字符串类型的属性，用来指定等待提示条的颜色。它的默认值是 **gray**。
- **hidesWhenStopped**，布尔类型的属性，用来指定当等待提示条不旋转时，是否显示在屏幕上。默认值是 **true**，表示当等待提示条不旋转时不显示。
- **onLayout function**，其作用与前面章节中讨论的相同，这里不再说明。
- **size**，字符串类型的属性，用来指定等待提示条的大小。它可以取值为 **small** 和 **large**。默认值是 **small**。

9.3.3 iOS 平台等待提示条例程

最简单的 iOS 平台等待提示条例程见代码 9-6。

代码 9-6：

```

'use strict';
var React = require('react-native');
var {
  AppRegistry,
  ActivityIndicatorIOS,
  View,
  StatusBarIOS
} = React;
var TimerMixin = require('react-timer-mixin');

```

```

var MovingBar = React.createClass({

mixins: [TimerMixin],
  getInitialState() {
    return {
      animating: true
    };
  },
  componentDidMount() {
    if ( StatusBarIOS !== null ) StatusBarIOS.setHidden(true);
    this.setInterval(
      () => {
        this.setState({animating: !this.state.animating});
      },
      2000
    );
  },
  render() {
    return (
      <View>
        <View style={{margin:10}}>
          <ActivityIndicatorIOS/>
        </View>
        <View style={{margin:10}}>
          <ActivityIndicatorIOS
            animating={this.state.animating}
            size="large"/>
        </View>
        <View style={{margin:10}}>
          <ActivityIndicatorIOS
            animating={this.state.animating}
            hidesWhenStopped={false}
            size="large"/>
        </View>
      </View>
    );
  },
});
AppRegistry.registerComponent('Project19', () => MovingBar);

```

代码 9-6 运行效果如图 9-4 所示。



图 9-4 iOS 手机等待提示条效果

在图 9-4 中，第一行的等待提示条会一直旋转并显示；第二行的等待提示条每次会旋转 2 秒钟时间，然后消失 2 秒钟时间，等待提示条只在旋转时才显示；第三行的等待提示条每次会旋转

2 秒钟时间，然后静止 2 秒钟时间，无论是旋转还是静止时等待提示条都会显示在屏幕上。

9.4 Switch 组件

Switch 组件是移动应用开发中经常使用到的开关组件。React Native 框架提供了 Switch 组件用于显示开关。

9.4.1 Switch 组件样式设置

Switch 组件支持所有的 View 组件的样式设置，开发者可以将其视为一个 box，使用 flexbox 的各种样式设置和 View 的样式设置。

9.4.2 Switch 其他属性

- disabled，布尔类型的属性，它的默认值为 false。当它为 true 时，移动应用使用者不能通过触摸开关组件来改变开关。
- onChange，函数类型的属性，开发者通过这个回调函数来检测用户改变开关组件的事件。
- Value，布尔类型的属性，用来设置开关组件的值。其使用方法见代码 9-7。
- 以下三个属性为 iOS 平台独有的属性：
- iosonTintColor，字符串类型的属性，用来设置开关打开（value 设为 true 时）时的背景色。它的默认值是 green（绿色）。
- iosthumbTintColor，字符串类型的属性，用来设置开关的颜色。它的默认值是 white（白色）。
- iostintColor，字符串类型的属性，用来设置开关被关闭时的颜色。它的默认值是 white（白色）。

9.4.3 Switch 组件的使用

最简单的 Switch 组件例程见代码 9-7。

代码 9-7：

```
'use strict';
var React = require('react-native');
var { AppRegistry, Switch, View, StatusBarIOS } = React;
var SwitchExample = React.createClass({
  getInitialState() {
    return {
      aSwitch: true
    };
  },
  componentDidMount() {
    if ( StatusBarIOS !== null ) StatusBarIOS.setHidden(true);
  },
```

```
aSwitchChanged:function() {  
  this.setState(()=>{  
    return{  
      aSwitch:!this.state.aSwitch  
    };  
  });  
},  
render() {  
  return (  
    <View >  
      <Switch style={{margin:20}}  
        onChange={this.aSwitchChanged}  
        value={this.state.aSwitch}/>  
      <Switch style={{margin:20}}  
        value={!this.state.aSwitch}/>  
    </View>  
  );  
},  
});  
AppRegistry.registerComponent('Project19', () => SwitchExample);
```

如图 9-5 和图 9-6 所示分别是代码 9-7 在 iOS 和 Android 平台上运行的效果。



图 9-5 iOS 手机开关组件效果

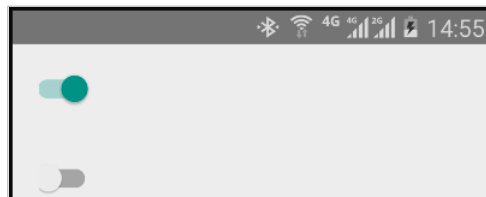


图 9-6 Android 手机开关组件效果

第 10 章

导航组件

为了让读者早些了解 React Native 框架的强大的 UI 能力，在第 3 章和第 5 章我们已经介绍了一些 Navigator 组件的用法。本章将详细讨论 Navigator 组件。

读者可能已经注意到还有一个导航组件名称为 NavigatorIOS。它也是一个实现导航功能的组件，但相比 Navigator 组件，它有以下几个缺陷：

- 轻量、受限的 API 设置，使其相对 Navigator 来说不太方便定制。
- 由开源社区主导开发 —— React Native 的官方团队并没有在自己的应用中使用该组件。这样导致目前有一些积压的 bug，而且没有人负责处理。
- 封装了 UIKit，因此和其他原生应用的表现完全一致。依赖 Objective-C 和 JavaScript，因此只能使用苹果开发好的动画和行为，并且这个组件只支持 iOS。
- 默认包含一个导航栏，这个导航栏不是 React Native 视图组件，因此只能稍微修改样式。

考虑到上述不足，因此不建议读者在 React Native 开发中使用 NavigatorIOS 组件。

10.1 导航组件的属性

10.1.1 回调函数

开发者可以通过指定 Navigator 组件的 `configureScene` 属性来定制场景切换时的动画效果。这是一个回调函数类型的属性，它接收场景切换的数据，然后返回开发者希望的场景切换动画效果。

`configureScene` 指定的回调函数被执行时，会收到一个导航路径参数，开发者可以根据导航路径中的各信息（由开发者在要求切换场景时提供）来决定场景切换使用何种效果。返回值可以是：`PushFromRight`、`FloatFromRight`、`FloatFromLeft`、`FloatFromBottom`、`FloatFromBottomAndroid`、`FadeAndroid`、`HorizontalSwipeJump`、`VerticalUpSwipeJump`、`VerticalDownSwipeJump`。`configureScene` 属性的具体使用与效果将在例程 10-1 中展示。

`onDidFocus` 属性用来指定一个回调函数。当导航组件导入初始场景后，或者每一个新的场景切换完成时，这个回调函数将被调用。它可以接收一个保存有新场景路径信息的参数。

现在 React Native 不建议开发者使用 `onDidFocus` 属性，而是鼓励开发者使用 `navigationContext`。

`addListener('didfocus', callback)`事件监听器来实现相同的功能。

`onWillFocus` 属性用来指定一个回调函数。当导航组件准备进行场景切换前，这个回调函数将被调用。

现在 React Native 同样不建议开发者使用 `onWillFocus` 属性，而是鼓励开发者使用 `navigationContext.addListener('willfocus', callback)`事件监听器来实现相同的功能。

`renderScene`。

10.1.2 其他属性

`sceneStyle`, `style` 类型的属性。如果开发者提供了这个属性，指定的样式将被应用到每一个切换的场景中。

`initialRoute`, 类类型的属性。在 3.2 节中我们已经看到了它的用法。如果给 `Navigator` 组件提供了 `initialRouteStack` 属性，那么 `initialRoute` 必须是 `initialRouteStack` 中的一个元素；如果提供了 `initialRouteStack` 但没有提供 `initialRoute`，那么 `initialRouteStack` 中的最后一个元素将默认为 `initialRoute`。

`initialRouteStack`, 类类型的属性，用来在 `Navigator` 组件初加载时提供导航路径。如果没有向 `Navigator` 组件提供 `initialRoute` 属性，就必须提供 `initialRouteStack` 属性；如果提供了 `initialRoute` 但没有提供 `initialRouteStack`，那么 React Native 会生成一个只有 `initialRoute` 元素的数组作为 `initialRouteStack`。

`navigationBar`, 该属性返回一个可以渲染的节点，这个节点可以用作所有界面的通用导航栏。我们将在 10.3 节中详细讨论它的用法。

如果当前组件是由父 `Navigator` 组件导航而产生的，则可以对其指定 `navigator` 属性，用来执行组件导航相关操作。在 3.2 节中，我们已经看到了这种用法。

10.2 导航器

在本书 3.4 节中，我们已经讨论了导航器的 `push`、`pop` 和 `replace` 函数的使用。导航器用于各界面导航的函数有：

- `getCurrentRoutes()`函数，用来得到当前的路径列表；
- `jumpBack()`函数，退回到上一个界面而不卸载当前界面；
- `jumpForward()`函数，沿界面路径向前跳一个界面而不卸载当前界面；
- `jumpTo(route)`函数，跳转到某个界面而不卸载任何界面；
- `push(route)`函数，导航组件在路径列表最前端添加一个新的界面，并马上跳转至这个界面；
- `pop()`函数，导航器退回一个界面并卸载原界面；
- `replace(route)`函数，导航器将当前界面用一个新的界面替代；
- `replaceAtIndex(route, index)`函数，使用一个新的界面替代路径列表中的第 `index` 个界面(下

标从 0 开始)，但不改变当前显示界面；

- `replacePrevious(route)`函数，将当前导航路径的上一个界面使用指定的界面替代；
- `immediatelyResetRouteStack(routeStack)`函数，使用给定的路径列表替换当前的路径列表；
- `popToRoute(route)`函数，导航器将回退到指定的界面，并在这个过程中将回退过的界面都一一卸载
- `popToTop()`函数，导航器会回到界面路径列表中的第一个界面，并且卸载其他所有界面。

10.3 NavigationBar

使用 `Navigator` 组件时，我们可以为 `Navigator` 组件管理的所有界面指定一个导航栏，这个导航栏将显示在指定的位置。如果没有指定这个位置，导航栏将默认显示在手机屏幕的最上方。

在 5.5.1 节中，我们已经讨论了如何实现自定义导航栏。自定义导航栏与使用 `NavigationBar` 实现的导航栏有什么区别呢？

从本质上说，`Navigator` 组件的导航栏是三个显示区域，开发者可以在这三个显示区域中显示任何 `React Native` 组件，如文字、图片、按钮、输入框等。开发者需要能：

- 设置三个区域的大小；
- 控制在这三个区域中显示的内容；
- 如果三个区域中有列表或者输入框，要能够控制用户按下按钮或者输入文字后的业务逻辑。

当开发者决定使用 `NavigationBar` 来进行界面导航时，大部分应用界面的导航栏都具有相同的格式（同样大小的按钮、标题栏等），只是按钮的图片或者标题栏中的文字各有不同。如果各应用界面的导航栏有不同的格式，这些导航的元素就应当在各个界面中被单独实现，而不是使用 `NavigationBar` 来实现。

当大部分应用界面的导航栏有相同的格式时，相比 5.5.1 节中的自定义导航栏，使用 `NavigationBar` 实现导航栏能让程序代码更精练、集中，应用逻辑更好处理。

但在实际应用中，仍然有个别界面的导航栏有特殊需求，不能套用统一的导航栏格式，`NavigationBar` 也支持这种需求，只是实现代码会复杂些。

给 `Navigator` 组件指定导航栏的示例如下：

```
navigationBar={
  <Navigator.NavigationBar routeMapper={ NavigationBarRouteMapper } />
}
```

其中的 `Navigator.NavigationBar` 是一个可显示的 `React Native` 组件，它必须有一个 `routeMapper` 属性。

开发者必须将一个对象指定给 `routeMapper` 属性。这个对象可以有三个成员变量：`LeftButton`、`RightButton` 和 `Title`。其中，`Title` 成员变量必须要有，其他两个视开发者需要来提供。这三个成员

变量要求都是函数类型的，Navigator 组件渲染导航栏时，使用这三个函数的返回值渲染导航栏的对应区域。

每个函数可以接收 4 个参数。示例如下：

```
LeftButton:function( route, navigator, index, navState )
```

在三个成员函数返回的可渲染节点的样式中设置三个区域的大小。这三个函数返回的可渲染节点就是三个区域中显示的内容。

不同的页面需要控制这三个区域中显示不同的内容，开发者需要将不同页面待显示的不同内容（文字、图片）通过 route 传入这三个函数中，然后这三个函数从 route 的成员变量中取出传入的供显示的不同内容，最后渲染显示。

对按钮或输入框的处理，通常都需要调用父组件的函数，这就需要将这个父组件的函数以某种方式传入 routeMapper 属性中。开发者无法直接给 routeMapper 属性再传值，但可以放在 route 中，由 Navigator 组件在渲染时交给 routeMapper 属性。而 route 中的成员变量，都是由开发者提供的，并且对每个事件只能提供一个回调函数（准确地说，是最近一次提供的回调函数会覆盖上一次提供的回调函数。这也正是 NavigationBar 组件目前使用不多的原因。我们将在例程中看到这一点。

例程 10-1 综合展示了 configureScene 的各种效果，以及 NavigationBar 的使用方法。

例程 10-1：

```
'use strict';
var React = require('react-native');
var PixelRatio = require('PixelRatio');
let pixelRatio = PixelRatio.get();
var {
  AppRegistry, StyleSheet, Navigator, Image, View, TouchableOpacity, Text
} = React;
var Image1 = require('./image/image1.jpg'); //开发者需要准备两张图片
var Image2 = require('./image/image2.jpg'); //以便于观看场景切换动画效果
var Image1Displayer = React.createClass({ //建立第一个显示组件
  changeScene: function() {
    //因为场景切换发生在一个修改状态机变量的过程中，不能在切换过程中再次调用 setState
    //修改状态机变量，所以在这里通过一个回调函数在场景切换之前通知父组件修改状态机变量
    this.props.callback();
    this.props.navigator.push({ //切换场景
      name: "2",
      UIIndex:this.props.UIIndex, //这个 Index 被显示在导航栏上，说明是第几个界面
      cbForLeftButton:this.props.cbForLeftButton //左导航栏按钮事件处理函数
    });
  },
  render: function() {
    return ( //Image1Displayer 显示一张图片和一段文字，文字是上一次场景切换效果的名称
      <View style={styles.container}>
        <TouchableOpacity onPress={this.changeScene}>
          <Image style={styles.imageStyle}
            source={Image1}/>
        </TouchableOpacity>
      </View>
    );
  }
});
```

```

        <Text style={styles.textStyle}>
            {this.props.textPrompt}
        </Text>
    </View>
    );
}
});
//Image2Displayer 基本上与 Image1Displayer 相同,只是显示另一张图片
//制作两个基本相同的组件只是为了展示导航组件的各种特性
var Image2Displayer = React.createClass({
    changeScene: function() {
        this.props.callback();
        this.props.navigator.push({
            name: "1",
            UIIndex: this.props.UIIndex,
            cbForLeftButton: this.props.cbForLeftButton
        });
    },
    render: function() {
        return (
            <View style={styles.container}>
                <TouchableOpacity onPress={this.changeScene}>
                    <Image style={styles.imageStyle}
                        source={Image2}/>
                </TouchableOpacity>
                <Text style={styles.textStyle}>
                    {this.props.textPrompt}
                </Text>
            </View>
        );
    }
});
var styles = StyleSheet.create({
    container: {
        flex: 1,
        justifyContent: 'center',
        alignItems: 'center',
    },
    imageStyle: {          //显示图片用的样式
        width: 1080 / pixelRatio / 2,
        height: 1920 / pixelRatio / 2
    },
    textStyle: {          //显示文字用的样式
        fontSize: 30,
        top: 50,
        left: 5,
    },
    buttonStyle: {
        fontSize: 20,
        margin: 10,
        backgroundColor: 'grey',
        width: 70
    },
    titleStyle: {          //导航栏中央区域文字显示样式
        fontSize: 20,

```

```

        margin: 10,
        left: 10,
        textAlign: 'center'
    }
  });
let NavigationBarRouteMapper = {      //定义设置导航栏的变量
  LeftButton: function(route, navigator, index, navState) {      //定义左侧区域如何显示
    let pString;
    if (route.textForLeftButton !== undefined) pString = route.textForLeftButton;
    else pString = '上一个';
    if (index > 0) {
      return ( <Text style = { styles.buttonStyle }
        onPress = { () => {
          try {
            route.cbForLeftButton(route.UIIndex);
            navigator.jumpBack();
          } catch (error) {
            //用户回退到头后还企图回退，这里 catch 的 error 不需要处理
          }
        }
      } >
        {pString}
      </Text>
    );
    }
    else {
      return (
        <Text style={ [styles.buttonStyle, {color: 'red'}] } >
          {pString}
        </Text>
      );
    }
  },
  Title: function(route, navigator, index, navState) {      //定义导航栏中部区域如何显示
    return (
      <Text style={styles.titleStyle}>
        第{route.UIIndex}个界面      //界面序号可以动态改变
      </Text>
    )
  },
  RightButton: function(route, navigator, index, navState) {      //定义右侧区域如何显示
    if (navState.sceneConfigStack.length === index + 1) {
      return (
        <Text style={ [styles.buttonStyle, {color: 'red'}] } >
          下一个
        </Text>
      )
    }
    return ( <Text style = {styles.buttonStyle}
      onPress = {() => {
        if (navState.sceneConfigStack.length === index + 1) {
          console.log('Can not jump forward.');
```

```

        }
        }>
        下一个
    </Text>
  )
}
};
var NaviExample = React.createClass({
  UIIndex:0,
  touchtime: 0,
  switchSceneStyle: Navigator.SceneConfigs.PushFromRight, //成员变量，用来指定切换效果
  getInitialState: function() {
    return {
      textPrompt: ''
    }; //状态机变量，用来在子组件中显示切换效果名称
  },
  callbackforLeftButton: function(aNumber) {
    console.log('call back function received number:'+aNumber);
  },
  changeStateVarBeforeRoute: function() { //子组件用这个函数通知父组件准备切换场景
    this.touchtime++; //点击次数成员变量加 1
    let newTextPrompt;
    switch (this.touchtime % 9) { //通过点击次数模 9 来达到每次切换效果的不同
      case 0: //按模 9 后的不同结果设置不同的值
        newTextPrompt='PushFromRight';
        this.switchSceneStyle = Navigator.SceneConfigs.PushFromRight;
        break;
      case 1:
        newTextPrompt='FloatFromRight';
        this.switchSceneStyle = Navigator.SceneConfigs.FloatFromRight;
        break;
      case 2:
        newTextPrompt='FloatFromLeft';
        this.switchSceneStyle = Navigator.SceneConfigs.FloatFromLeft;
        break;
      case 3:
        newTextPrompt='FloatFromBottom';
        this.switchSceneStyle = Navigator.SceneConfigs.FloatFromBottom;
        break;
      case 4:
        newTextPrompt='FloatFromBottomAndroid';
        this.switchSceneStyle = Navigator.SceneConfigs.FloatFromBottomAndroid;
        break;
      case 5:
        newTextPrompt='FadeAndroid';
        this.switchSceneStyle = Navigator.SceneConfigs.FadeAndroid;
        break;
      case 6:
        newTextPrompt='HorizontalSwipeJump';
        this.switchSceneStyle = Navigator.SceneConfigs.HorizontalSwipeJump;
        break;
      case 7:
        newTextPrompt='VerticalUpSwipeJump';
        this.switchSceneStyle = Navigator.SceneConfigs.VerticalUpSwipeJump;
        break;
    }
  }
});

```

```

        case 8:
            newTextPrompt='VerticalDownSwipeJump';
            this.switchSceneStyle = Navigator.SceneConfigs.VerticalDownSwipeJump;
        }
        this.setState(() => {          //更改文字提示的状态机变量
            return {
                textPrompt: newTextPrompt
            };
        });
    },
    configureScene: function(route) {
        return this.switchSceneStyle;          //设置当前切换使用何种效果
    },
    renderScene: function(router, navigator) {
        switch (router.name) {
            case "1":
                return <Image1Displayer navigator={navigator}
                    textForLeftButton='新文字'
                    //在这里还需要将导航栏左侧按钮回调函数与界面序号再赋值一次, 否则会出现异常
                    //如果要深究原因, 需要看 Navigator 组件和 NavigationBar 组件源码
                    cbForLeftButton={this.callbackforLeftButton}
                    UIIndex={this.touchtime}
                    textPrompt={this.state.textPrompt}
                    callback={this.changeStateVarBeforeRoute}/>;
            case "2":
                return <Image2Displayer navigator={navigator}
                    //在这里还需要将导航栏左侧按钮回调函数与界面序号再赋值一次, 否则会出现异常
                    cbForLeftButton={this.callbackforLeftButton}
                    UIIndex={this.touchtime}
                    textPrompt={this.state.textPrompt}
                    callback={this.changeStateVarBeforeRoute}/>;
        }
    },
    render: function() {
        return ( <Navigator initialRoute = { {
            name: '1',
            UIIndex:0,
            cbForLeftButton: this.callbackforLeftButton,
            textForLeftButton: '新文字'
        } }
            configureScene = {this.configureScene}
            //定义导航栏组件如何实现
            navigationBar = {<Navigator.NavigationBar routeMapper={ NavigationBarRouteMapper } />}
            renderScene = {this.renderScene}>
            </Navigator>
        );
    }
});
AppRegistry.registerComponent('Project19', () => NaviExample);

```

例程 10-1 的运行结果是，用户每点击一次手机屏幕上的图片，就会使用一种切换模式进行页面切换，同时切换模式的名称也显示在手机上，方便读者理解每一种切换效果。在界面的上方有导航栏，导航栏中间的文字标明是第几个界面，数字会跟随用户的点击而改变。用户点击左侧按

钮可以回到上一个界面（再点击再回，直到回到第 0 个界面），点击右侧按钮可以去下一个界面（如果它存在的话）。

如果需要使用 `NavigationBar`，则可以仔细研究此例程。

最后总结一下使用 `NavigationBar` 实现导航栏的特点：如果只是简单地在一系列预设好的界面路径中按序切换，那么使用 `NavigationBar` 是最合适不过了。如果导航栏功能复杂一些，建议开发者还是使用 5.5.1 节中讨论的自定义导航栏。

第 11 章

手势识别

11.1 PanResponder API

PanResponder API 将多种触摸行为协调成一个手势。它可以很方便地追踪一个单点触摸的后续发展，也可以用于识别简单的多点触摸手势。

React Native 框架底层的手势响应系统提供了响应处理器，PanResponder API 将这些手势响应处理器再次进行封装，以便于开发者更容易对手势进行处理，更容易预测用户的手势。对每一个手势响应处理器，PanResponder API 除了为其提供代表触摸行为的原生事件外，还提供了一个新的手势状态对象用来详细描述手势的状态。

React Native 开发小组把这个 API 的名字起得比较有意思。从字面上理解，这个名字的意思是对用户手指按压屏幕的行为产生反应。但从它的工作原理来说，称为触摸监视器更合适。

11.2 监视器

PanResponder API 的基本思想是：监视屏幕上指定大小、位置的矩形区域，当用手指按压这个区域中的某点后，开发者会收到这个事件；当按压后拖动手指时，开发者会收到手势引发的各类事件；当手指离开这个矩形区域时，开发者也会收到相应的事件。

需要注意的是，开发者可以任意指定监视矩形区域的大小，但在这个区域里，只有第一个按下的事件会上报和继续监视处理。如果将监视区域设为整个屏幕或者半个屏幕，当把一个手指放在屏幕上时，开发者可以监视到手指被放在屏幕上、移动和最终离开屏幕的整个过程。但如果当一个手指放在屏幕上，还没有离开时，又一个手指放在了手机屏幕的监视区域里，那么开发者将对第二个手指的各种触摸事件一无所知。

开发者可以在屏幕上指定多个监视矩形区域，但不能同时监视多个区域中的不同触摸事件，也就无法利用 PanResponder API 来处理多点触摸事件了。当第一个触摸事件发生后并没有结束时，开发者无法通过 PanResponder API 获取此期间发生的其他触摸事件。

PanResponder API 与 React Native 框架的 View 组件，或者从 View 组件继承来的组件联系非常紧密，它监测的矩形区域的大小和位置都是由与它挂接的组件来指定的。

如果开发者不希望监视区域改变当前手机 UI 的显示，则可以使用 View 组件来指定监视区域，

同时将这个 View 组件的背景色设为全透明。

如果开发者希望监视区域在当前手机 UI 上有可视效果,则可以与某个 View(或者某个 Image) 紧密联系,开发者可以针对这个 View 轻松实现手势的视觉效果。比如检测到某事件后,按业务逻辑改变该 View 的位置(实现跟随用户手指效果)、透明度、颜色、大小、圆角率、子组件各种属性等。

不论监视区域是否改变当前手机 UI 的显示,都会阻止被监视区域覆盖的组件接收触摸事件。比如监视区域盖住了一个按钮,那么就无法通过按这个按钮来触发其对应的事件,开发者只能在监视器的事件处理函数中对触摸行为进行处理。

利用 PanResponder API 实现监视器有以下几个步骤。

11.2.1 指定监视区域

为了监视(下一步移动)一个区域,开发者需要准备一个 View 或者一个从 View 组件扩展而来的组件。就像平时定义组件一样,为它定义一个 Style,然后将它放在组件的 render 返回 JSX 代码中。

当需要监视多个区域时,一定要注意不能让任意两个监视区域有任何重叠。当两个监视区域有重叠时,会导致两个监视区域的监视器都不能正常工作。

为了实现触摸事件的可视化效果,开发者有时会动态改变监视区域在手机屏幕上的位置,这时就更要注意不能与其他监视区域产生任何重叠。

11.2.2 定义监视器相关变量

与监视器相关的变量有:

- 指向监视器的变量。这个变量必须存在。
- 用来指向监视器监视区域的变量。可以不定义这个变量;但当触摸发生需要给用户视觉上的反馈时,有这个变量可以很容易实现反馈。
- 用来记录监视区域左上角顶点坐标的两个数值变量。可以不定义这个变量;但当触摸发生需要给用户视觉上的反馈时,有这个变量可以很容易实现反馈。
- 上一次触摸点的横、纵坐标变量。可以不定义这两个变量;但有这两个变量,便于分析、处理触摸事件。

11.2.3 准备监视器的事件处理函数

监视器可能会上报给开发者的事件共有 13 个,它们是:

```
onMoveShouldSetPanResponder
onMoveShouldSetPanResponderCapture
onStartShouldSetPanResponder
onStartShouldSetPanResponderCapture
onPanResponderReject
```

```
onPanResponderGrant
onPanResponderStart
onPanResponderEnd
onPanResponderRelease
onPanResponderMove
onPanResponderTerminate
onPanResponderTerminationRequest
onShouldBlockNativeResponder
```

开发者需要从中挑选出自己关心的事件，为这些事件实现事件处理函数。这些事件将在 11.3 节中详细讨论。

11.2.4 建立监视器

利用 PanResponder API 提供的静态函数 create，开发者建立起一个监视器。在建立监视器时，需要指明开发者准备了哪些事件处理函数并把这些函数与对应事件挂接。建立监视器的代码通常类似于代码 11-1。

代码 11-1:

```
this.watcher = PanResponder.create({
  onStartShouldSetPanResponder: this._handleStartShouldSetPanResponder,
  onMoveShouldSetPanResponder: this._handleMoveShouldSetPanResponder,
  onPanResponderGrant: this._handlePanResponderGrant,
  onPanResponderMove: this._handlePanResponderMove,
  onPanResponderEnd: this._handlePanResponderEnd,
});
```

11.2.5 将监视器与监视区域挂接

最后一步是将监视器与监视区域挂接。假设通过代码 11-1 实现了一个可以被 this.watcher 引用的监视器，那么挂接的代码就类似于代码 11-2。

代码 11-2:

```
<View ref={(aViewID) => { this.watchingAreaViewID = aViewID; }}
  style={styles.watchingAreaStyle}
  {... this.watcher.panHandlers}/>
```

第一句是将系统生成的用来引用监视区域的组件 ID 记录在成员变量中，以便开发者在后续处理流程中通过这个成员变量对监视区域进行操作。

第三句是正式挂接的语句。它通过 JSX 的延展属性语法，将 this.watcher.panHandlers 中的所有属性传递给监视区域组件。用一条简单的语句完成了很多复杂的操作。

11.3 监视事件的生命周期

当手指按压在监视区域时，通过一系列计算，React Native 框架取出被按压区域的中心点，然后认为这个中心点被按压。

如果没有特殊说明，在单次点击的生命周期中被回调函数都会收到两个参数，其中一个原是

生事件（`nativeEvent`）；另一个是手势状态（`gestureState`）。

原生事件有以下成员变量：

- `changedTouches`——在上一次事件之后，所有发生变化的触摸事件的数组集合（即上一次事件后，所有移动过的触摸点）；
- `identifier`——触摸点的 ID；
- `location`——触摸点相对于父元素的横坐标；
- `location`——触摸点相对于父元素的纵坐标；
- `pageX`——触摸点相对于根元素的横坐标；
- `pageY`——触摸点相对于根元素的纵坐标；
- `target`——触摸点所在元素的 ID；
- `timestamp`——触摸事件的时间戳，可用于移动速度的计算；
- `touches`——当前屏幕上的所有触摸点的集合。

虽然原生事件定义得很完备，但在 `PanResponder` API 中，在所有事件上报来的 `nativeEvent` 中只有 `target` 字段有值，并且开发者通常不关心这个值。因此在事件处理函数中，通常都不处理原生事件参数。

手势状态有以下成员变量：

- `stateID`——触摸状态的 ID。在屏幕上至少有一个触摸点的情况下，这个 ID 会一直有效；
- `moveX`——最近一次移动时的屏幕横坐标；
- `moveY`——最近一次移动时的屏幕纵坐标；
- `x0`——当响应器产生时的屏幕坐标；
- `y0`——当响应器产生时的屏幕坐标；
- `dx`——从触摸操作开始的累计横向路程；
- `dy`——从触摸操作开始的累计纵向路程；
- `vx`——当前的横向移动速度；
- `vy`——当前的纵向移动速度；
- `numberActiveTouches`——当前在屏幕上有效触摸点的数量。

在事件处理函数中，主要分析手势状态参数中的各成员变量值，并按业务逻辑进行处理。我们将在后续的例程中看到这些内容。

11.3.1 单次点击事件的生命周期

单次点击事件的完整生命周期按发生时间顺序排列如下：

1. `onStartShouldSetPanResponderCapture`

`onStartShouldSetPanResponderCapture` 事件让开发者来决定当手指与监视区域接触时，是否设置开始捕捉这次触摸的相关事件。开发者从事件处理函数收到的参数中可以得到手势状态 ID、触

摸点所在元素的 ID 和当前活跃的触摸点数。注意,此时开发者不能够知道按压点的具体坐标信息,只知道按压点在监控区域内。

如果有多个手指按压在同一个监视区域,那么只要有一个手指还没有离开监视区域,这次手势就没有结束,手势状态的 ID 也就不会被改变。

如果 `onStartShouldSetPanResponderCapture` 返回 `true`,将跳过第 2 步,直接进入第 3 步,否则进入第 2 步。

2. `onStartShouldSetPanResponder`

`onStartShouldSetPanResponder` 事件让开发者再有一次机会决定当手指与监视区域接触时,是否设置开始捕捉这次触摸的相关事件。开发者从事件处理函数收到的参数中可以得到手势状态 ID、触摸点所在元素的 ID 和当前活跃的触摸点数。

如果有多个手指按压在同一个监视区域,那么只要有一个手指还没有离开监视区域,这次手势就没有结束,手势状态的 ID 也就不会被改变。注意,此时开发者不能够知道按压点的具体坐标信息,只知道按压点在监控区域内。

如果 `onStartShouldSetPanResponder` 返回 `true`,将进入第 3 步,否则不会收到第 3 个至第 7 个事件,直到这个手势状态结束。

提示:读者可能会疑惑,第 1 个事件和第 2 个事件感觉很相同,有什么区别吗,或者说第 1 步与第 2 步之间发生了什么事件吗?答案是没有区别,第 1 步与第 2 步之间也没有任何事件发生。

如果开发者需要捕捉每一个点击事件,则将第 1 步的返回值直接设为 `true`,而跳过第 2 步。

如果开发者需要视情况决定是否需要捕捉点击事件,则可以不设置第 1 步的处理函数,而使用系统默认处理(默认返回 `false`)函数,然后在第 2 步视应用的业务逻辑决定是否捕捉本次点击事件。

3. `onPanResponderGrant`

`onPanResponderGrant` 事件通知开发者监视器开始工作了。开发者可以从事件处理函数收到的参数中得到手势状态 ID、触摸点所在元素的 ID、触摸点的横坐标触摸点的纵坐标和当前活跃的触摸点数。

此函数无须返回值。

4. `onShouldBlockNativeResponder`

`onShouldBlockNativeResponder` 事件让开发者决定当前组件是否要阻止原生组件成为触摸事件的响应器。返回 `true` 时表示阻止,返回 `false` 时表示不阻止。截至 React Native 0.17.0 版本,这个函数只对 Android 平台有效。开发者通常不需要设置这个处理函数,不设置时,默认的处理行

为是阻止。

这个事件处理函数将不会收到任何参数。

5. onPanResponderStart

`onPanResponderStart` 事件通知开发者一个触摸事件已经正式被监视。开发者可以从事件处理函数收到的参数中得到手势状态 ID、触摸点所在元素的 ID、触摸点的横坐标、触摸点的纵坐标和当前活跃的触摸点数。

6. onPanResponderEnd

`onPanResponderEnd` 事件通知开发者一个触摸事件已经结束。开发者可以从事件处理函数收到的参数中得到手势状态 ID、触摸点所在元素的 ID、触摸点的横坐标、触摸点的纵坐标和当前活跃的触摸点数。

7. onPanResponderRelease

`onPanResponderRelease` 事件通知开发者一个监视器已经被释放。开发者可以从事件处理函数收到的参数中得到手势状态 ID、触摸点所在元素的 ID、触摸点的横坐标、触摸点的纵坐标和当前活跃的触摸点数。

这个事件与第 6 个事件没有区别，原因同上。

上述的 7 个事件，在一次点击屏幕中会上报一次（如果开发者实现并挂接了相应的事件处理函数），并且只会上报一次。

11.3.2 单次点击事件处理

11.3.1 节描述了单次点击事件的整个生命周期，从描述中我们可以知道很多事件开发者是不需要处理的。通常开发者处理事件的流程是：

（1）实现对第 2 个事件 `onStartShouldSetPanResponder` 的处理函数，按业务逻辑判断是否监视本次触摸事件。

（2）实现对第 5 个事件 `onPanResponderStart` 的处理函数，将这个事件视为点击事件的开始点。

（3）实现对第 6 个事件 `onPanResponderEnd` 的处理函数，将这个事件视为点击事件的结束点。

11.3.3 移动手势事件的生命周期

当手指按压在监视区域后，没有马上离开，而是在屏幕上移动时，就形成了移动手势。移动手势事件的生命周期有两种。

11.3.3.1 点击事件演化为移动手势事件

在 11.3.1 节中，点击事件进展到第 5 步后，手指开始移动，这时开发者会不定时收到一个

onPanResponderMove 事件。在 PanResponder API 中，onPanResponderMove 事件的触发规则是：

- 当向某个方向移动停止时会上报 onPanResponderMove 事件；
- 当向某个方向移动了足够长的距离后会上报 onPanResponderMove 事件；
- 如果手势在移动中停下来但又没有离开屏幕，那么在手势停止期间都不会上报事件，直到下一次移动开始或者手指离开屏幕。

按照上述的触发规则，当手势移动速度足够快时，onPanResponderMove 事件可以达到每 25 毫秒甚至更短上报一次。

onPanResponderMove 会按触发规则上报给处理函数，直到手指离开监视区域。此时将进入 11.3.1 节中描述的第 6 步和第 7 步。

11.3.3.2 专注移动手势处理

如果开发者不需要处理点击事件，只希望处理移动手势，那么 11.3.1 节中讨论的第 1 个事件和第 2 个事件都返回 false，就可以不处理点击事件。这时如果手势从点击发展为移动，那么移动手势的生命周期按时间顺序排列如下：

1. onMoveShouldSetPanResponderCapture

onMoveShouldSetPanResponderCapture 事件将会上报给对应的处理函数，让开发者决定是否捕捉这次手势移动的后续事件。如果返回 true，将跳过第 2 步，直接进入第 3 步；如果返回 false，则进入第 2 步

开发者可以从事件处理函数收到的 gestureState 参数中得到它的所有成员变量，这些成员变量中记载着此次移动手势的各项数据。

2. onMoveShouldSetPanResponder

onMoveShouldSetPanResponder 事件将会上报给对应的处理函数，再次让开发者决定是否捕捉这次手势移动的后续事件。如果返回 true，将进入第 3 步。如果处理函数返回 false，而移动手势还在继续，那么不定时长后，又会开始一个新的移动手势事件生命周期。新的生命周期开始的触发条件，请参照 11.3.3.1 节中描述的 onPanResponderMove 事件的触发条件。

开发者可以从事件处理函数收到的 gestureState 参数中得到它的所有成员变量，这些成员变量中记载着此次移动手势的各项数据。

提示：第 1 个事件和第 2 个事件依然没有区别，第 1 步与第 2 步之间也没有任何事件发生。

如果开发者需要捕捉每一个移动手势事件，就把第 1 步的返回值设为 true，跳过第 2 步。

如果需要视情况来判断，开发者可以不设置第 1 步的处理函数，而使用系统默认处理（默认返回 false）函数，然后在第 2 步视业务逻辑决定是否捕捉本次移动手势事件。

3. onPanResponderGrant

onPanResponderGrant 事件通知开发者监视器开始工作了。开发者可以从事件处理函数收到的 gestureState 参数中得到它的所有成员变量，这些成员变量中记载着此次移动手势的各项数据。

这个事件的处理函数无须返回任何值。

4. onShouldBlockNativeResponder

onShouldBlockNativeResponder 事件让开发者决定当前组件是否要阻止原生组件成为触摸事件的响应器。返回 true 时表示阻止，返回 false 时表示不阻止。截至 React Native 0.17.0 版本，这个函数只对 Android 平台有效。开发者通常不需要设置这个处理函数，不设置时，默认的处理行为是阻止。

这个事件的处理函数将不会收到任何参数。

第 4 个事件被触发后，将进入 11.3.3.1 节中描述的 onPanResponderMove 事件触发阶段。

11.3.4 监视器异常事件

PanResponder API 提供了 onPanResponderReject 事件用来上报开发者要求监视某个区域被拒绝。

PanResponder API 提供了 onPanResponderTerminationRequest 事件用来上报开发者监视器被要求终止。这个事件处理函数返回 false 表示不同意终止，或者不处理这个事件。

PanResponder API 提供了 onPanResponderTerminate 事件通知开发者监视器被异常终止。终止的原因可能是另一个组件已经成为新的响应者，也有可能是其他程序（比如电话呼入程序）抢占了手机屏幕。

11.4 手势识别处理例程

本节将讨论如何将手势识别处理应用到实践中，让其与其他 React Native 组件结合，完成各种手势识别功能。

11.4.1 单点手势——点击、拖动选择百分比参数

该百分比选择器例程允许用户通过点击、拖动来为某个参数（比如音量大小）选择百分比。例程的运行效果（上半部分截图，下面都是空白）如图 11-1 所示。

百分比选择器例程见代码 11-3。

代码 11-3:

```
'use strict';
var Dimensions = require('Dimensions');
```



图 11-1 百分比选择器例程运行效果

```

var totalWidth = Dimensions.get('window').width;
var React = require('react-native');
var {AppRegistry, PanResponder, StyleSheet, View, ProgressViewIOS, Text} = React;
var RatioChooser = React.createClass({
  watcher: null, //成员变量用来保存监视器
  getInitialState: function() {
    return {progress: 0}; //状态机变量用来保存选择的百分比参数
  },
  componentWillMount: function() {
    this.watcher = PanResponder.create({ //建立监视器
      onStartShouldSetPanResponder: () => true, //这个事件处理函数直接返回 true
      onPanResponderGrant: this._onPanResponderGrant, //我们只关心按下和移动两个
      onPanResponderMove: this._onPanResponderMove, //事件, 只挂接这两个事件处理函数
    });
  },
  _onPanResponderGrant: function(e: Object, gestureState: Object) {
    let touchPointX = gestureState.x0; //获取触摸点的横坐标, 不获取纵坐标, 是因为不需要知道
    let progress;
    if ( touchPointX < 20 ) progress = 0; //如果按压点超过起点, 百分比值按 0 算
    else {
      if ( touchPointX > (totalWidth-40) ) progress = 1; //如果按压点超过终点, 按 1 算
      else progress = (touchPointX - 20) / (totalWidth-40); //计算出对应的百分比值
    }
    this.setState({progress}); //将计算出的百分比值交给状态机变量
  },
  _onPanResponderMove: function(e: Object, gestureState: Object) {
    let touchPointX = gestureState.moveX; //获取移动时触摸点的横坐标
    let progress;
    if ( touchPointX < 20 ) progress = 0; //如果按压点超过起点, 百分比值按 0 算
    else {
      if ( touchPointX > (totalWidth-40) ) progress = 1; //如果按压点超过终点, 按 1 算
      else progress = (touchPointX - 20) / (totalWidth-40); //计算出对应的百分比值
    }
    this.setState({progress}); //将计算出的百分比值交给状态机变量
  },
  render: function() {
    return (
      <View style={styles.container}>
        <ProgressViewIOS style={styles.ProgressViewStyle}
          progress={this.state.progress}/>
        <Text style={styles.textStyle}>
          你选择了{Math.round(this.state.progress*100)}%
        </Text>
        <View style={styles.touchViewStyle}
          {...this.watcher.panHandlers}/> //将监视器与这个 View 挂接
      </View>
    );
  },
});
var styles = StyleSheet.create({
  ProgressViewStyle: {
    width: totalWidth-40, //组件不要顶边显示, 不美观
    left: 20,
    top: 50,
  },
});

```

```

    },
    container: {
      flex: 1
    },
    touchViewStyle: { //这是监视器挂接的监视区域的样式定义
      width: totalWidth-20, //监视区域比进度条宽，便于用户点击
      height: 40, //监视区域比进度条高，便于用户点击
      backgroundColor: 'transparent', //设置背景色为透明，不影响它遮盖的组件的显示
      position: 'absolute',
      left: 10,
      top: 32,
    },
    textStyle: {
      fontSize: 30,
      left: 20,
      top: 70
    }
  }
});
AppRegistry.registerComponent('Project19', () => RatioChooser);
;

```

在该例程的 `render` 函数中，可以不定义透明的 `View` 作为监视区域，而是把监视器直接与 `ProgressViewIOS` 组件挂接。这样实现的代码见代码 11-4。

代码 11-4:

```

render: function() {
  return (
    <View style={styles.container}>
      <ProgressViewIOS style={styles.ProgressViewStyle}
        progress={this.state.progress}
        {...this.watcher.panHandlers}/> //将监视器与 ProgressViewIOS 组件挂接
    </View>
  );
},
});

```

对该例程按代码 11-4 修改后，执行是没有问题的，用户也可以进行点击、拖动选择。只是因为 `ProgressViewIOS` 的区域太小，用户会发现很难点击到监视区域从而改变百分比。这也是为什么在代码 11-3 中要使用一个 `View` 作为监视区域的原因。

11.4.2 单点手势——带导槽的滑动来电接听或拒接界面

滑动来电接听或拒接例程运行效果如图 11-2 所示（手机屏幕上半部分截图，下面是空白）。

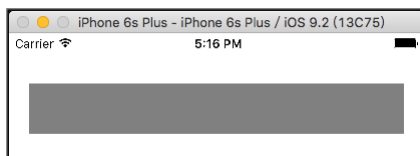


图 11-2 滑动来电接听或拒接例程运行效果

当手指按在图中灰色导槽的最左端时，导槽最左侧会变为绿色，并且绿色会跟随手指向右延

伸或者向左缩回，当松开手指时，导槽会恢复灰色。

当手指按在图中灰色导槽的最右端时，导槽最右侧会变为红色，并且红色会跟随手指向左延伸或者向右缩回，当松开手指时，导槽会恢复灰色。

如果用户的手指按在导槽的中间区域，导槽不会有任何反应。

滑动来电接听或拒接例程见代码 11-5。

代码 11-5，index.ios.js 或者 index.android.js:

```
'use strict';
var Dimensions = require('Dimensions');
var totalWidth = Dimensions.get('window').width;
let miniViewWidth = (totalWidth-40)/10;
var React = require('react-native');
var {AppRegistry, PanResponder, StyleSheet, View} = React;
var IncomingCall = React.createClass({
  watcher: null, //用来记录监视器
  startFromLeft:true, //用来判断最先按下的是最左端还是最右端
  moveNeedhandle:false, //用来判断监测到的移动事件是否需要处理

  newViewColors:['grey','grey','grey','grey','grey','grey','grey','grey','grey','grey'],
  getInitialState:function() {
    return {viewColors:this.newViewColors}; //记录 10 个子 View 的颜色
  },
  componentWillMount: function() {
    this.watcher = PanResponder.create({ //建立监视器
      onStartShouldSetPanResponder: ()=>true, //这个事件处理函数直接返回 true
      onPanResponderGrant: this._onPanResponderGrant, //我们关心按下、移动和松手三个
      onPanResponderMove: this._onPanResponderMove, //事件，挂接三个处理函数
      onPanResponderEnd: this._onPanResponderEnd,
    });
  },
  _onPanResponderGrant: function(e: Object, gestureState: Object) {
    let touchPointX=gestureState.x0; //取得按压点的横坐标
    if ( touchPointX < 20 ) return; //按压点向左超过导槽，不处理
    if ( touchPointX > totalWidth-20 ) return; //按压点向右超过导槽，不处理
    if ( (touchPointX > (20+miniViewWidth)) && (touchPointX < (totalWidth-20-miniViewWidth)) ) return; //按压点位不在导槽最左端或者最右端，不处理
    this.moveNeedhandle = true; //从现在起，需要处理监测到的手势移动事件
    if ( touchPointX < (20+miniViewWidth)) {
      this.startFromLeft=true; //初始按压点在最左端
      this.newViewColors[0] = 'green'; //最左端第一个子 View 变为绿色
    }
    else {
      this.startFromLeft=false; //初始按压点在最右端
      this.newViewColors[9] = 'red'; //最右端第一个子 View 变为红色
    }
    this.setState(()=>{
      return {
        viewColors:this.newViewColors //要求重新渲染界面
      };
    });
  }
});
```

```

    },
    _onPanResponderMove: function(e: Object, gestureState: Object) {
        if ( !this.moveNeedhandle ) return; //如果不需要处理手势移动事件就直接返回
        let touchPointX=gestureState.moveX; //获取当前按压点的横坐标
        if( this.startFromLeft ) { //初始的按压点在导槽的最左端
            //计算需要将多少个子 View 改变为绿色
            let howManyBlock = Math.round((touchPointX-20)/miniViewWidth);
            if ( howManyBlock === 10 ) {
                //手指移动到导槽最右端, 确定选择接听电话, 界面跳转
            }
            //下面两条语句用来改变各子 View 的颜色
            for(let index=1;index<howManyBlock;index++) this.newViewColors[index]='green';
            for(let index=howManyBlock+1;index<10;index++) this.newViewColors[index]='grey';
        }
        else { //初始的按压点在导槽的最右端
            //计算需要将多少个子 View 改变为红色
            let howManyBlock = Math.round((totalWidth-20-touchPointX)/miniViewWidth);
            if ( howManyBlock === 10 ) {
                //手指移动到导槽最左端, 确定选择拒绝电话, 界面跳转
            }
            //下面的语句用来改变各子 View 的颜色
            let index;
            let backup = howManyBlock;
            for( index=8;howManyBlock>0;index-- ) {
                howManyBlock--;
                this.newViewColors[index]='red';
            }
            for(;index>=0;index--) this.newViewColors[index]='grey';
        }
        //处理完毕, 要求重新渲染界面
        this.setState(()=>{
            return {
                viewColors:this.newViewColors
            };
        });
    },
    _onPanResponderEnd:function(e: Object, gestureState: Object) {
        //松开手指, 让各变量回到初始状态
        for(let index=0;index<10;index++) this.newViewColors[index]='gray';
        this.moveNeedhandle=false;
        this.setState(()=>{
            return {
                viewColors:this.newViewColors
            };
        });
    },
    render: function() {
        return (
            <View style={styles.container}>
                <View style={styles.barViewStyle}
                    {...this.watcher.panHandlers}> //挂接监视器
                    //下面是 10 个子 View, 颜色由状态机变量决定
                    <View style={styles.miniViewStyle,{backgroundColor: this.state.viewColors[0]}}/>
                    <View style={styles.miniViewStyle,{backgroundColor: this.state.viewColors[1]}}/>

```

```

      <View style={ [styles.miniViewStyle, {backgroundColor: this.state.viewColors[2]}] } />
      <View style={ [styles.miniViewStyle, {backgroundColor: this.state.viewColors[3]}] } />
      <View style={ [styles.miniViewStyle, {backgroundColor: this.state.viewColors[4]}] } />
      <View style={ [styles.miniViewStyle, {backgroundColor: this.state.viewColors[5]}] } />
      <View style={ [styles.miniViewStyle, {backgroundColor: this.state.viewColors[6]}] } />
      <View style={ [styles.miniViewStyle, {backgroundColor: this.state.viewColors[7]}] } />
      <View style={ [styles.miniViewStyle, {backgroundColor: this.state.viewColors[8]}] } />
      <View style={ [styles.miniViewStyle, {backgroundColor: this.state.viewColors[9]}] } />
    </View>
  </View>
);
},
});
var styles = StyleSheet.create({
  barViewStyle: {
    width: totalWidth-40,
    height: 50,
    left: 20,
    top: 50,
    flexDirection: 'row'
  },
  container: {
    flex: 1
  },
  miniViewStyle: {
    width: miniViewWidth,
    height: 50
  }
});
AppRegistry.registerComponent('Project19', () => IncomingCall);

```

这只是一个简单的示例程序。还有两个可以改进的地方：①导槽两端应当有文字提示；②导槽只被分为 10 等分，还是有些太粗糙。

该例程只是让读者明白如何实现这个功能。真正要商用的话，不仅仅是导槽两端应当有文字提示，也不仅仅要求把导槽分得再细一些，而应当是导槽的所有组成部分都是一张张精心设计的图片，随着用户的手势图片被动态替换，形成动画效果。还是那句老话：需要一个好的美工。

11.4.3 单点手势——滑动解锁屏幕界面

如图 11-3 所示是滑动解锁屏幕界面（或者滑动接听电话）例程运行效果。这个例程与上一个例程的不同点在于，在这个例程中，滑动的导槽不会有视觉上的变化，而是手指按压的滑块跟随手指移动。也就是说，按压的监视区域是随着手指移动而变化的。

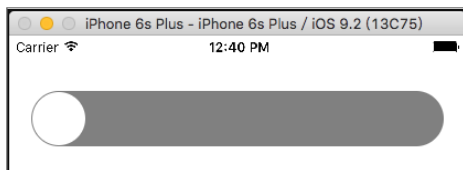


图 11-3 滑动解锁屏幕界面例程运行效果

滑动解锁屏幕界面例程见代码 11-6。

代码 11-6:

```
'use strict';
var Dimensions = require('Dimensions');
var totalWidth = Dimensions.get('window').width;
var React = require('react-native');
var {AppRegistry, PanResponder, StyleSheet, View } = React;
var IncomingCall = React.createClass({
  watcher: null,          //记录监视器的成员变量
  startX:0,              //记录按下时触摸点偏移值的成员变量
  getInitialState:function() {
    return {leftPoint:1};    //记录滑块偏移值
  },
  componentWillMount: function() {
    this.watcher = PanResponder.create({
      onStartShouldSetPanResponder: ()=>true,
      onPanResponderGrant: this._onPanResponderGrant,
      onPanResponderMove: this._onPanResponderMove,
      onPanResponderEnd: this._onPanResponderEnd,
    });
  },
  _onPanResponderGrant: function(e: Object, gestureState: Object) {
    this.startX = gestureState.x0;    //按住滑块，记录偏移值
  },
  _onPanResponderMove: function(e: Object, gestureState: Object) {
    let leftPoint;                //移动滑块，计算滑块偏移值
    if ( gestureState.moveX < this.startX ) {
      leftPoint = 1;              //滑块向左偏移到尽头，不继续偏移
    }
    else {
      if ( gestureState.moveX > totalWidth-42-48+this.startX ) {
        leftPoint = totalWidth - 42-48;    //需要改变滑块的显示位置
      }
      else {
        leftPoint=gestureState.moveX - this.startX; //滑块向右偏移到尽头，不继续偏移
        //解锁屏幕，跳转界面（或者接通电话，跳转界面）
      }
    }
    this.setState(()=>{
      return{leftPoint};          //请求重新渲染界面
    });
  },
  _onPanResponderEnd:function(e: Object, gestureState: Object) {
    let leftPoint=1;              //松开手指，滑块恢复默认值
    this.setState(()=>{
      return{leftPoint};
    });
  },
  render: function() {
    return (
      <View style={styles.container}>
        <View style={styles.barViewStyle}>          //下一条语句，滑块的位置由状态机变量决定
          <View style={[styles.buttonViewStyle,{left: this.state.leftPoint}]}
            {...this.watcher.panHandlers}/>          //挂接监视器
        </View>
      </View>
    );
  }
});
```

```

        </View>
      </View>
    );
  },
});
var styles = StyleSheet.create({
  barViewStyle: {
    width: totalWidth-40,
    height:50,
    backgroundColor:'grey',
    borderRadius:25,      //定义圆角风格
    left: 20,
    top: 50,
    flexDirection: 'row'
  },
  container: {
    flex: 1
  },
  buttonViewStyle: {
    width: 48,            //滑块的宽、高等于导槽的高减 2
    height: 48,
    borderRadius:24,      //定义圆角风格
    backgroundColor:'white',
    left:1,              //从导槽的(1,1)坐标开始显示
    top:1
  }
});
AppRegistry.registerComponent('Project19', () => IncomingCall);

```

这个例程也可以很方便地更改为两端按压，以选择接听来电或者拒绝来电的界面。在这里大致说一下修改思路。

(1) 增加一个监视区域，初始时位置位于导槽的最右端，每个监视区域各使用一张图片表明一个按下时是准备接听，另一个按下时是准备拒绝来电。

(2) 当某个监视区域被按下时，这个监视区域的图片高亮显示，同时另一个监视区域的图片被改变为全透明的，并且被移动出导槽以给被按下的滑块腾让位置（**注意，两个监视区域的位置不能有任何重叠！**）。

(3) 监控滑块是否被滑动到另一头顶端。如果将滑块滑动到另一头顶端，那么是选择确认，跳转界面，执行选择。

(4) 如果滑动滑块到中途松手，则需要将两个监视区域恢复到初始值，以便被再次选择。

(5) 如果再结合 11.4.2 节中的例程，在滑动过程中，导槽显示也有视觉上的变化，效果将更好。

11.4.4 单点手势——单点任意方向拉动选择界面

如图 11-4 所示是某些手机来电时的用户选择界面。用户可以按压绿色或者红色圆形按钮，然后移动手指，这时会有一个透明的绿色（或者红色）的圆形跟随手指移动。当手指移动到离圆心

足够远时，界面确认用户已经做出了选择，然后按照选择接听或者拒绝电话。



图 11-4 某些手机来电时的用户选择界面

代码 11-7 展示了如何实现其中一个选择的各种效果。

在代码 11-7 中，圆心使用了第 7 章日记例程中的“悲伤心情图标”，当手指按上它后，圆心变为“高兴心情图标”。当手指向任意方向移动时，一个同心圆会跟随手指扩展，直到扩展到足够大。程序判定用户已经确定了选择或者松手，回到初始状态。

代码 11-7:

```
'use strict';
var Dimensions = require('Dimensions');
let totalWidth = Dimensions.get('window').width;
let totalHeight = Dimensions.get('window').height;
let startHeight = totalHeight - totalWidth/2 + 5; //这是监视区域上沿的高度
let PixelRatio = require('PixelRatio');
let pixelRatio = PixelRatio.get();
let centerX = 5 + (totalWidth/2-10)/2; //这是监视区域中心点的横坐标
let centerY = startHeight + (totalWidth/2-10)/2; //这是监视区域中心点的纵坐标
var unTouchedIcon = require('./image/sad.jpg'); //当没有点击时显示的图片
var touchedIcon = require('./image/happy.jpg'); //当点击后图片切换为这一张
var React = require('react-native');
var {AppRegistry, PanResponder, StyleSheet, View, Image } = React;
var IncomingCall = React.createClass({
  watcher: null,
  getInitialState:function() {
    return {
      imageIcon: unTouchedIcon,
      width: (totalWidth/2-10)/4, //状态机变量用来控制扩展的圆形的宽度
      height: (totalWidth/2-10)/4, //状态机变量用来控制扩展的圆形的高度
      left: ((totalWidth/2 - 10) - ((totalWidth/2-10)/4))/2, //圆形的左起点
      top: ((totalWidth/2 - 10) - ((totalWidth/2-10)/4))/2, //圆形的上起点
      borderRadius: (totalWidth/2-10)/8 //状态机变量用来将正方形显示为圆形
    };
  },
  componentWillMount: function() {
    this.watcher = PanResponder.create({
      onStartShouldSetPanResponder: ()=>true,
      onPanResponderGrant: this._onPanResponderGrant,
      onPanResponderMove: this._onPanResponderMove,
```

```

        onPanResponderEnd: this._onPanResponderEnd,
      });
    },
    _onPanResponderGrant: function(e: Object, gestureState: Object) {
      //下面的语句计算按压点与监视区域中心点的距离
      let distance = this.caculateDistanse(gestureState.x0,gestureState.y0)
      if (distance < (totalWidth/2-10)/8 ) { //距离在范围内,说明按压了中心的图片
        this.dragStart = true; //允许监视移动手势的操作
        this.setState(()=>{
          return {imageIcon:touchedIcon}; //改变中心区域的显示图片
        });
      }
    },
    _onPanResponderMove: function(e: Object, gestureState: Object) {
      if ( !this.dragStart ) return; //如果没有按压到中心图标,不处理手势移动事件
      //下面的语句计算按压点与监视区域中心点的距离
      let distance = this.caculateDistanse( gestureState.moveX, gestureState.moveY);
      if ( distance < (totalWidth/2-10)/8 ) return; //手势没有移出出中心图标区域
      if ( distance > (totalWidth/2-10)/2 ) {
        //手势已经移动到允许的最大值,用户的选择被确认,跳转界面,执行指定的操作
      }
      let left = ((totalWidth/2 - 10) -((totalWidth/2-10)/4))/2-distance + (totalWidth/2-10)/8; //手势在合法移动中,计算与之对应的扩展圆形的各个显示参数
      let top = ((totalWidth/2 - 10) -((totalWidth/2-10)/4))/2-distance + (totalWidth/2-10)/8;
      let width = 2*distance;
      let height = 2*distance;
      let borderRadius=distance;
      //重新渲染界面,实现圆形扩展的动画效果
      this.setState(()=>{
        return {left,top,width,height,borderRadius}; //改变 5 个状态机变量
      });
    },
    _onPanResponderEnd:function(e: Object, gestureState: Object) {
      let imageIcon=unTouchedIcon; //手指脱离屏幕,将各项显示恢复为初始值
      let width=(totalWidth/2-10)/4;
      let height=(totalWidth/2-10)/4;
      let left= ((totalWidth/2 - 10) -((totalWidth/2-10)/4))/2;
      let top=((totalWidth/2 - 10) -((totalWidth/2-10)/4))/2;
      let borderRadius=(totalWidth/2-10)/8;
      this.dragStart = false;
      this.setState(()=>{
        return {imageIcon,left,top,width,height,borderRadius};
      });
    },
  },
  //此函数用来计算给定坐标与监控区域中心点的距离
  caculateDistanse: function( x,y ) {
    return Math.sqrt(Math.pow((x- centerX),2)+Math.pow((y- centerY),2));
  },
  render: function() {
    return (
      <View style={styles.container}>
        <View style={styles.transparentViewStyle}

```

```

        {...this.watcher.panHandlers}>           //挂载监视器
        <View
style={([styles.shadowViewStyle,{borderRadius:this.state.borderRadius,
left:this.state.left,top:this.state.top,width:this.state.width,height:this.state.he
ight }])}/>
        //通过样式的动态设置，实现扩展的圆形
        <Image style={styles.touchImageStyle}
            source={this.state.imageIcon}/>
        </View>
    </View>
    );
  },
});
var styles = StyleSheet.create({
  container: {
    flex: 1
  },
  touchImageStyle: {           //触摸区域中心点图片样式设置
    width: (totalWidth/2-10)/4,
    height: (totalWidth/2-10)/4,
    borderRadius: (totalWidth/2-10)/8,
    position: 'absolute',
    left: ((totalWidth/2 - 10) - ((totalWidth/2-10)/4))/2,
    top: ((totalWidth/2 - 10) - ((totalWidth/2-10)/4))/2,
    backgroundColor: 'grey'
  },
  shadowViewStyle: {           //扩展的圆形初始样式设置，因为与上一张图
    width: (totalWidth/2-10)/4,           //片大小一样，初始时会被图片遮盖住
    height: (totalWidth/2-10)/4,
    left: ((totalWidth/2 - 10) - ((totalWidth/2-10)/4))/2 ,
    top: ((totalWidth/2 - 10) - ((totalWidth/2-10)/4))/2 ,
    backgroundColor: 'blue',
    borderRadius: (totalWidth/2-10)/8,
    position: 'absolute'
  },
  transparentViewStyle: {       //监视区域样式设置
    width: totalWidth/2 - 10,
    height: totalWidth/2 - 10,
    backgroundColor: 'transparent',
    left: 5,
    top: startHeight
  }
});
AppRegistry.registerComponent('Project19', () => IncomingCall);

```

本例程中使用了“blue”作为扩展圆形的颜色，没有透明效果。真正商业开发时，让美工给一个 RGBA 值，就可以轻松实现半透明效果。

11.4.5 两点手势

截至 React Native 0.20.0 版本，PanResponder API 不支持在设备屏幕上建立两个监视区域，并同时监控两个监视区域中的触摸、移动事件。这个功能，还有待 React Native 开发社区对其进行完善。

第 12 章

网络

使用 React Native 框架开发时，可以使用 NetInfo API 来获取手机当前的各个网络状态。

React Native 框架在初始化项目时，在项目目录下安装了 node-fetch 包，开发者可以使用 node-fetch 包通过 HTTP 协议来获取网络侧的数据

12.1 获取网络状态

移动应用通常希望能用下面两种方式获取网络状态。

12.1.1 得到当前网络状态

代码 12-1 可以让开发者得到当前网络状态。

代码 12-1:

```
.....
NetInfo.fetch().done((status) => {
  console.log('Status: ' + status);
});
.....
```

请注意，获取网络状态是异步的，也就是说，发出获取网络状态的代码与得到并处理当前网络状态的代码是分开执行的，在它们执行过程中其他代码有可能会被执行。在代码 12-1 中使用了 JavaScript 的 Promise 机制，让这段代码看起来好像是同步执行的，比较有欺骗性。

12.1.1.1 Android 手机网络状态

为了获取 Android 手机网络状态，开发者需要先修改 React Native 项目目录下的 \android\app\src\main\AndroidManifest.xml 文件，它是 Android 项目的配置文件。在文件中加入：

```
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
```

修改完成后，需要在项目目录下重新运行“react-native run-android”命令。这个命令重新编译对应的 Android 项目安装包并安装到手机中。在这个过程中，如果服务器在运行，则会被停止。需要再次启动服务器。

修改、编译、安装、重启后，代码 12-1 打印出的 status 有可能为下列值：

- NONE——手机没有任何网络连接；
- BLUETOOTH——当前数据连接通过蓝牙协议执行；
- DUMMY——当前数据连接为假数据连接；
- ETHERNET——当前使用以太网数据连接；
- MOBILE——当前使用手机网络数据连接；
- MOBILE_DUN——当前使用拨号移动网络数据连接；
- MOBILE_HIPRI——当前使用高优先级移动网络数据连接；
- MOBILE_MMS——当前使用彩信移动网络数据连接；
- MOBILE_SUPL——当前使用安全用户面定位（SUPL）数据连接；
- VPN——当前使用虚拟网络连接，需要在 Android 5.0 以上；
- WIFI——当前使用 WiFi 数据连接；
- WIMAX——当前使用 WiMAX 数据连接；
- UNKNOWN——当前使用未知数据连接。

12.1.1.2 iPhone 手机网络状态

代码 12-1 在 iPhone 手机上运行时，打印出的 status 有可能为下列值：

- none ——当前没有网络连接；
- wifi ——当前手机使用 WiFi 网络连接，或者是一个 iOS 模拟器；
- cell ——当前手机使用 Edge、3G、WiMAX 或者 LTE 网络连接；
- unknown ——发生错误，网络状况不可知。

12.1.2 监听网络状态改变事件

在获取了网络状态后，开发者还可以通过 NetInfo API 提供的监听器，监听网络状态改变事件。这样当手机网络状态改变时，React Native 应用能马上收到通知。

设置监听器后，只有在网络状态改变时，才会收到通知。通常设置网络监听器的代码如代码 12-2 所示。

代码 12-2：

```
.....
removeListener:null,                                     //成员变量，用来卸载监听器
handleConnectivityChange:function(status) {
  console.log('status change: ' + status); //简单打印状态变化
},
componentWillMount:function() {
  NetInfo.fetch().done((reach) => {
    console.log('Initial: ' + reach);
  });
  this.removeListener = NetInfo.addEventListener('change',
this.handleConnectivityChange); //监听并记录卸载函数
```

```
    },
    componentWillUnmount:function() {
      this.removeListener();
    },
  },
  .....
  //卸载监听器
```

网络状态处理函数得到的网络状态字符串的含义，与 12.1.1 节中的网络状态字符串的含义一致。

12.1.3 简单判断是否有网络连接

NetInfo API 为开发者提供了 `isConnected` 函数用来判断当前手机是否有网络连接。

函数执行结果与获取网络状态一样是异步返回的。调用示例代码如代码 12-3 所示。

代码 12-3:

```
.....
NetInfo.isConnected.fetch().done((isConnected) => {
  console.log('First, is ' + (isConnected ? 'online' : 'offline'));
});
.....
```

12.1.4 判断当前连接是否收费

NetInfo API 为开发者提供了 `isConnectionExpensive` 函数用来判断当前网络连接是否是付费的。目前这个函数只为 Android 平台提供。

调用示例代码如代码 12-4 所示。

代码 12-4:

```
.....
NetInfo.isConnectionExpensive((boolValue) => {
  console.log('Connection is ' + (boolValue ? 'charged for.' : 'free of charge.'));
});
.....
```

12.2 通过 HTTP、HTTPS 与网络侧交换数据

通过 HTTP、HTTPS 协议与网络侧服务器交换数据是移动应用中常见的一种通信方式。React Native 集成了 `node-fetch` 包以支持开发者的这种需求。

学习本章需要部分 HTTP 协议相关知识。

12.2.1 发送请求

发送 HTTP 请求（或者 HTTPS 请求）有 6 个步骤。

12.2.1.1 确定并准备请求地址与协议

如果请求地址以“http://”开头,那么消息通过 HTTP 协议传输;如果请求地址以“https://”开头,那么消息通过 HTTPS 协议传输。为了便于修改这个地址,通常将这个地址放在一个变量中,而不是直接写在代码中。

示例代码片段:

```
let REQUEST_URL = 'http:// http://api.rottentomatoes.com/api/public/v1.0
/lists/movies/in_theaters.json ?apikey= 7waqfqbprs7pajbz28mqf6vz &page_limit=25';
```

12.2.1.2 确定请求中的 HTTP 方法

HTTP 请求(或者 HTTPS 请求)有 4 种方法,分别是“GET”、“PUT”、“POST”和“DELETE”。如果不指定,默认请求中的方法为 GET。

示例代码片段:

```
let map = {
  method: 'POST'
};
```

12.2.1.3 确定并准备请求中需要传输的消息头

接下来需要设置请求中需要传输的消息头。消息头分为两种,其中一种是协议规定的标准消息头;另一种是用户自定义消息头。

示例代码片段如下。

代码 12-5:

```
.....
let privateHeaders= {
  'Private-header1': 'value1',      //自定义消息头 1
  'Private-header2': 'value2',      //自定义消息头 2
  'Content-Type': " text/plain"    //设置消息体格式
  "User-Agent": 'testAgent'
}
map.headers=privateHeaders;          //加上定义的消息头
map.follow=20;                       //设置请求允许的最大重定向次数, 0 为不允许重定向
map.timeout=0;                       //设置超时时间, 0 为没有超时时间, 这个值在重定向时会被重置
map.size=0;                          //设置请求回应中的消息体最大允许长度, 0 为没有限制
.....
```

React Native 使用 HTTP 协议栈框架支持 gzip/deflate 格式编码,开发者不需要对此进行设置。在收到的回应中,消息体如果使用 gzip/deflate 格式编码,HTTP 协议栈框架会自动将消息体转为普通格式交给开发者。

设置消息头时需要注意如下事项:

- GET 方法不允许有消息体,因此不需要像代码 12-5 中第 4 行那样设置消息体类型。
- 对于其他方法,如果需要携带消息体,必须要像代码 12-5 中第 4 行那样设置消息体类型,

如果没有设置消息体类型却又携带了消息体，代码会报错。

- 除了消息体类型这个消息头，其他的定义的消息头都会把消息头字符串中的大写字母转为小写。
- 可以不设 User-Agent 消息头，这时 HTTP 协议框架会自己填上一个，值为“okhttp/2.5.0”。

12.2.1.4 确定请求中是否需要加入身份验证信息

在某些 HTTP 请求中需要加入身份验证信息。

从本质上讲，身份验证信息就是按 HTTP 协议相关条文，在一些约定好的 HTTP 消息头中填入身份认证信息。因此在 12.2.1.3 节中准备消息头的方法也适用于身份验证消息头。在进行身份验证时，有可能需要两次 HTTP 请求来完成，那么开发者按照验证协议发送两次 HTTP 请求即可。

12.2.1.5 确定请求是否需要携带消息体

如果请求需要携带消息体，那么其实现代码很简单，示例如下：

```
map.body='This is a message body for test';
```

12.2.1.6 发出消息

将上述的 5 步合起来，就可以得到发送 HTTP（或者 HTTPS）请求的代码片段。示例如代码 12-6 所示。

代码 12-6：

```
.....
let REQUEST_URL = 'http:// http://api.rottentomatoes.com/api/public/v1.0/lists
/movies/in_theaters.json ?apikey= 7waqfqbprs7pajbz28mqf6vz &page_limit=25';
let map = {method: 'POST'};
let privateHeaders={
  'Private-header1': 'value1',
  'Private-header2': 'value2',
  "Content-Type": "text/plain",
  "User-Agent": "testAgent"
};
map.headers=privateHeaders;
map.body='This is a message body for test';
map.follow=20;
map.timeout=0;
map.size=0;
fetch(REQUEST_URL, map ).then(
  (result) => {
    .....//接收到回应后的处理
  }
).catch((error)=> {
  console.log('error:'+error);
});
.....
```

代码 12-6 运行后，通过抓包软件抓到它发出的 POST 请求，如图 12-1 所示。

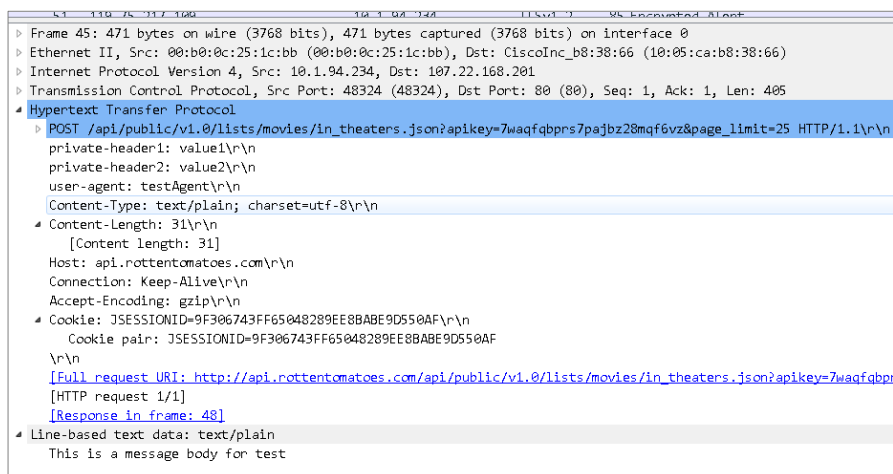


图 12-1 HTTP POST 请求抓包解析图

12.2.2 接收响应

当操作成功后，开发者得到一个存储有回应数据的对象。在代码 12-6 中，用 `result` 这个形式参数指向这个对象，开发者需要使用到这个对象的成员变量和成员函数。

提示：代码 12-6 发出的消息为 POST 请求，这是为了配合发出有消息体的消息。读者在进入 12.2.2 节学习之前，请把代码 12-6 改为发出 GET 消息，同时去掉消息体。

12.2.2.1 获取回应状态

- `type`，字符类型的成员变量，用来记录请求的类型，但当前还没有支持获取这个值；
- `url`，字符类型的成员变量，用来记录请求的地址，但当前还没有支持获取这个值；
- `status`，数值类型的成员变量，用来记录回应码；
- `ok`，布尔类型的成员变量，用来记录请求是否成功；
- `statusText`，字符类型的成员变量，用来保存字符串格式的请求响应码，但当前还没有支持获取这个值；

12.2.2.2 获取回应中的消息头与对应值

`Headers` 是对象类型的成员变量，用来保存请求回应中的 HTTP 消息头与对应值。它只有一个固定的子成员变量 `map`，`map` 又是一个对象类型的成员变量。

当请求的回应码是 200 时，`headers` 有三个固定的子成员：

- `headers.map.okhttp-selected-protocol`，这个值通常是 `["http/1.1"]`。注意，它是只有一个元素的字符串类型数组。
- `headers.map.okhttp-received-millis`，它是只有一个元素的字符串类型数组，记录收到回应的的时间值。例如：`["1453875008303"]`。

- `headers.map.okhttp-sent-millis`，它是只有一个元素的字符串类型数组，记录发现请求时的时间值。例如：`["1453875007767"]`。

因为 HTTP 是开放协议，因此使用 HTTP 协议的实体可以在 HTTP 协议中加入自定义的 HTTP 消息头与对应值，所以 HTTP 消息头是不可穷举的。

当开发者需要获取所收到回应中的特定 HTTP 消息头的对应值时，以特定 HTTP 消息头的全小写为键名从 `map` 中获得。例如：移动应用和网络服务器定了一个私有 HTTP 头，名称是 `Private-Header-Value`，那么在 React Native 代码中，开发者使用 `headers.map.private-header-value` 就可以得到这个消息头的对应值。

除了两个消息头，React Native 会将其余所有的 HTTP 消息头都通过 `headers.map`。“对应名称”提供给开发者。不提供的两个消息头是 `Content-Encoding` 和 `Content-Length`。这两个消息头与回应的消息体有关，开发者可以直接通过对象的 `text` 成员函数得到消息体，因此不需要关心这两个消息头。

12.2.2.3 获取回应中携带的消息体

- `_bodyInit`，字符类型的成员变量，这个字符串记载着请求回应的消息体。
- `json`，成员函数，当请求回应的消息体是 JSON 格式时，可以直接使用这个函数得到一个对应的对象。

12.2.2.4 复制回应消息

- `clone`，成员函数，用来复制一个记录请求回应数据的对象。它的实现如代码 12-7 所示。

代码 12-7：

```
clone () {  
    return new Response(this._bodyInit, {  
        status: this.status,  
        statusText: this.statusText,  
        headers: new Headers(this.headers),  
        url: this.url});  
}
```

12.3 在 React Native 开发中使用 AJAX 技术

AJAX 是 Asynchronous JavaScript And XML（异步 JavaScript 和 XML）的缩写，它可以被看成是一个系统设计与开发技术的合集。AJAX 支持网站与应用程序，它使用实时数据更新网页界面而无须页面刷新。该功能创建了一种更为流畅且更具桌面风格的用户体验。通过在后台与服务端进行少量数据交换，AJAX 可以使网页实现异步更新。这意味着可以在不重新加载整个网页的情况下，对网页的某部分进行更新。而传统的网页（不使用 AJAX）如果需要更新内容，则必须重载整个网页。

AJAX 的核心是 JavaScript 对象 `XMLHttpRequest`。`XMLHttpRequest` 是一套通过 HTTP 协议

传送或者接收 XML 及其他数据的 API。

目前 AJAX 还处于百家争鸣、百花齐放的阶段，W3C 启动了 AJAX 标准与规范的制定工作，但进展缓慢。

AJAX 开发技术不在本书的讨论范围内，因为它足以写出一本比本书更厚的书。同时因为标准的缺失，使用 AJAX 开发需要客户端（移动应用）与网络服务器程序紧密配合调测。

如果开发者以前的移动应用中使用到了 AJAX 技术，那么可以在 React Native 中继续使用 AJAX 技术。JavaScript 有一整套 XMLHttpRequest API，而 React Native 已经集成了这套 API。想要继续了解该技术的读者，只需要去看在 JavaScript 下如何实现 AJAX 开发的书籍即可。

第 13 章

网页浏览器、音视频媒体播放

React Native 框架提供了 WebView 组件供开发者在 UI 界面上指定一个区域用来渲染手机系统原生提供的网页浏览视图。

13.1 WebView 组件样式设置

WebView 组件可以使用 View 组件的所有样式设置。它自己没有其他的样式设置。

13.2 WebView 组件其他属性

WebView 组件可以接受所有的 View 组件的属性。除此之外，它还有自己独有的属性。

13.2.1 非回调函数属性

- `automaticallyAdjustContentInsets`，布尔类型的属性，用来打开或者关闭 WebView 组件自动调整网页内容功能。
- `contentInset`，类变量类型的属性，它可以接收一个类型为 `{top: number, left: number, bottom: number, right: number}` 的对象，用来定义 WebView 组件中显示内容距 WebView 四边的距离。截至 React Native 0.18.0 版本，它只在 iOS 平台上有效。
- `html`，字符串类型的属性，用来在 WebView 组件中显示指定的 HTML 字符串。我们将在 13.3.2 节中详细讨论它的用法。
- `injectedJavaScript`，字符串类型的属性，用来指定当网页开始加载时需要运行的 JavaScript 代码。
- `startInLoadingState`，布尔类型的属性，用来指定 WebView 组件在刚开始加载时的 Loading 状态（在页面位置显示等待提示条），等待网页读取完成后显示。
- `url`，字符串类型的属性，用来指定 WebView 组件加载的网址。在 React Native 0.19.0 版本后，开发者不应当再使用这个属性，而是使用 `source` 属性。为了保持与原有的应用代码的兼容，这个属性被保留。
- `source`，对象类型的属性。它期待的对象结构是如下两个结构之一：

```
{  
  uri: string,           //用来指定 WebView 组件加载的网址
```

```

method: string,      //用来指定加载的方法，除非服务器配合支持，否则不要设置
headers: object,     //用来指定加载时的 HTTP 消息头，除非服务器配合支持，否则不要设置
body: string         //用来指定加载的 HTTP 消息体，除非服务器配合支持，否则不要设置
},

{
  html: string,      //用来指定直接加载的 HTML 页面格式
  baseUrl: string    //用来将需要加载的文件路径写成相对于项目的相对路径
},

```

13.2.2 回调函数属性

- `onError` 回调函数，用来指定 `WebView` 组件加载网页发生错误时的处理函数。
- `onLoad` 回调函数，用来指定加载网页结束时的处理函数。
- `onLoadEnd` 回调函数，在网页加载结束或者网页加载失败时都会被调用。
- `onLoadStart` 回调函数，在 `WebView` 组件开始加载网页时将被调用。
- `onNavigationStateChange` 回调函数，在导航状态改变时将被调用。
- `renderError` 回调函数，需要返回一个 `View` 组件，当 `WebView` 组件加载网页出错时，将在手机屏幕上渲染这个 `View` 组件与它的所有子组件。
- `renderLoading` 回调函数，可以返回一个加载状态指示器。当 `WebView` 组件加载网页时，这个加载状态指示器将被渲染在 `WebView` 组件界面上直到加载结束。

13.2.3 平台独有属性

下面对 `WebView` 组件各平台独有的属性进行讨论。

13.2.3.1 iOS 平台独有属性

- `allowsInlineMediaPlayback`，布尔类型的属性，用来决定 HTML 5 网页中的视频是在网页中播放还是使用原生的全屏视频播放器播放。它的默认值是 `false`，表示使用原生的全屏视频播放器播放。如果希望视频在网页中播放，不仅要将这个属性设置为 `true`，而且 HTML 5 网页中相应的视频元素必须包含 `webkit-playsinline` 属性。
- `bounces`，布尔类型的属性，用来决定当网页在 `WebView` 组件中被拖动到尽头时，是否有弹动的 UI 显示特效。
- `onShouldStartLoadWithRequest`，回调函数类型的属性。如果设置了这个回调函数，在 `WebView` 访问的网页中触发的任何 JavaScript 发起的请求都会交给这个回调函数。回调函数分析 JavaScript 发起的请求，然后返回 `true` 或者 `false` 以决定是否执行 JavaScript 发起的请求。
- `scalesPageToFit`，布尔类型的属性，用来决定网页是否适配当前 `WebView` 的大小，以及是否能放大或者缩小网页。
- `scrollEnabled`，布尔类型的属性，用来决定是否允许当前网页上下滚动。

- `decelerationRate`, 字符串类型的属性, 它可以在 `normal` 和 `fast` 中取一值, 用来决定 `WebView` 在用户停止划动动作后, 页面减速直至不再移动的减速速度是快速的还是普通的。

13.2.3.2 Android 平台独有属性

- `domStorageEnabled`, 布尔类型的属性, 用来决定是否允许 `DOM` 在本机存储。
- `javaScriptEnabled`, 布尔类型的属性, 用来决定是否允许运行网页中的 `JavaScript` 脚本。在 `iOS` 平台上, 原生网页浏览器始终允许运行 `JavaScript` 脚本。

13.2.4 WebView 组件成员函数

- `onLoadingStart` 函数, 供开发者调用 `WebView` 组件的 `onLoadStart` 回调函数 (如果它被提供)。
- `onLoadingError` 函数, 供开发者调用 `WebView` 组件的 `onLoadError` 回调函数 (如果它被提供)。
- `onLoadingFinish` 函数, 供开发者调用 `WebView` 组件的 `onLoadEnd` 回调函数 (如果它被提供)。
- `reload` 函数, 供开发者调用以重新加载当前页面。

13.3 网页浏览器使用例程

13.3.1 浏览网页例程

代码 13-1 示范了如何使用 `WebView` 组件访问新浪新闻网页。

代码 13-1, `index.android.js` 或者 `index.ios.js`:

```
'use strict';
var React = require('react-native');
var {
  AppRegistry, StyleSheet, Text, TextInput,
  TouchableOpacity, View, WebView, StatusBar
} = React;
var Project20 = React.createClass({
  inputURL: '', //成员变量, 用来记录用户输入的网址
  getInitialState: function() {
    return {
      source: {
        uri: 'http://news.sina.com.cn' //设置打开网址
      },
      status: 'No Page Loaded', //默认状态栏文字
      backButtonEnabled: false, //后退按钮是否可按
      forwardButtonEnabled: false, //前进按钮是否可按
    };
  },
  onNavigationStateChange: function(navState) {
    //可以通过 navState.url 获取当前加载的网页元素
```

```

//可以通过 navState.loading 获取加载是否完成
this.setState({
  backButtonEnabled: navState.canGoBack,          //更新三个值用于显示
  forwardButtonEnabled: navState.canGoForward,
  status: navState.title
});
},
render: function() {
  return (
    <View style={[styles.container]}>
      <StatusBar hidden={true}/>                //隐藏手机状态栏
      <View style={[styles.addressBarRow]}>      //首行 View 定义
        <TouchableOpacity                      //后退按钮定义
          onPress={this.goBack}
          style={this.state.backButtonEnabled ? styles.navButton : styles.disabledButton}>
          <Text>
            {'<'}
          </Text>
        </TouchableOpacity>
        <TouchableOpacity                      //前进按钮定义
          onPress={this.goForward}
          style={this.state.forwardButtonEnabled ? styles.navButton : styles.
disabledButton}>
          <Text>
            {'>'}
          </Text>
        </TouchableOpacity>
        <TextInput ref='urlInputRef'            //网址输入框定义
          autoCapitalize="none"
          defaultValue={this.state.url}
          onSubmitEditing={this.pressGoButton}
          onChangeText={ (newText)=>{this.inputURL=newText}}
          clearButtonMode="while-editing"
          style={styles.addressBarTextInput}/>
        <TouchableOpacity onPress={this.pressGoButton}> //Go 按钮定义
          <View style={styles.goButton}>
            <Text>
              Go!
            </Text>
          </View>
        </TouchableOpacity>
      </View>
      <WebView ref='webViewRef'                //网络浏览器定义
        automaticallyAdjustContentInsets={false}
        style={styles.webView}
        source={this.state.source}
        javaScriptEnabled={true}
        domStorageEnabled={true}
        onNavigationStateChange={this.onNavigationStateChange}
        startInLoadingState={true}/>
      <View style={styles.statusBar}>          //网络浏览器状态栏定义
        <Text style={styles.statusBarText}>{this.state.status}</Text>
      </View>
    </View>
  );
}

```

```

    );
  },
  goBack: function() {          //让 WebView 组件退回
    this.refs.webViewRef.goBack();
  },
  goForward: function() {       //让 WebView 组件前进
    this.refs.webViewRef.goForward();
  },
  pressGoButton: function() {   //打开用户输入的网址
    var uri = this.inputURL.toLowerCase();
    if (uri === this.state.source.uri) { //当网址与当前网页相同时，直接让 WebView 组件重载
      this.refs.webViewRef.reload();
    }
    else {
      let source = {};
      source.uri = uri;
      this.setState({source});
    }
  },
});
var styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: 'blue',
  },
  addressBarRow: {              //第一行(导航按钮、网址输入框)容器的样式定义
    flexDirection: 'row',
    padding: 8,
  },
  webView: {                   //网页浏览器组件样式定义
    backgroundColor: 'white',
    height: 350,
  },
  addressBarTextInput: {       //网址输入框样式定义
    backgroundColor: 'white',
    borderColor: 'transparent',
    borderRadius: 3,
    borderWidth: 1,
    height: 24,
    paddingLeft: 10,
    paddingTop: 3,
    paddingBottom: 3,
    flex: 1,
    fontSize: 14,
  },
  navButton: {                 //导航按钮可用时样式定义
    width: 20,
    padding: 3,
    marginRight: 3,
    alignItems: 'center',
    justifyContent: 'center',
    backgroundColor: 'white',
    borderColor: 'transparent',
    borderRadius: 3,
  },
});

```



```

disabledButton: {                                //导航按钮不可用时样式定义
  width: 20,
  padding: 3,
  marginRight: 3,
  alignItems: 'center',
  justifyContent: 'center',
  backgroundColor: 'grey',
  borderColor: 'transparent',
  borderRadius: 3,
},
goButton: {                                       //Go 按钮样式定义
  height: 24,
  padding: 3,
  marginLeft: 8,
  alignItems: 'center',
  backgroundColor: 'white',
  borderColor: 'transparent',
  borderRadius: 3,
  alignSelf: 'stretch',
},
statusBar: {                                     //网页浏览器状态栏样式定义
  flexDirection: 'row',
  alignItems: 'center',
  paddingLeft: 5,
  height: 22,
},
statusBarText: {                                //网页浏览器状态栏文字样式定义
  color: 'white',
  fontSize: 13,
}
});
AppRegistry.registerComponent('Project20', () => Project20);

```

13.3.2 加载本地网页例程

在移动应用开发中，原生 **WebView** 组件经常用来展示应用的帮助、关于页面信息。这些页面通常由文字和图片构成，美工经常是直接把构成网页的相关 **HTML** 文件和图片文件提供给开发者，开发者直接使用原生 **WebView** 组件加载网页呈现给用户。

从 **React Native 0.20.0** 版本开始，**React Native** 框架的 **WebView** 组件可以直接加载位于项目目录下的 **HTML** 文件，并且支持多页面文件之间的链接跳转。但在 **0.20.0** 版本上，还有一点局限，就是开发者必须把所有需要使用到的 **HTML** 文件和图片文件都存放在项目根目录下，不能存放在子目录中。

我们这里提供两个简单的 **HTML** 文件。注意，这两个文件必须要以 **UTF-8** 编码格式保存。

代码 13-2, A.html:

```

<!doctype html>
<html>
  <head>
    <meta charset="utf-8">

```

```
    <title>A 页内容</title>
  </head>
  <body>
    <div><a href="b.html" >点击进入 B 页面</a></div>
    <p>A 页内容</p>
  </body>
</html>
```

代码 13-3, B.html:

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>B 页内容</title>
  </head>
  <body>
    <div><a href="A.html" >点击进入 A 页面</a></div>
    <p>B 页内容</p>
  </body>
</html>
```

读取并显示这两个帮助 HTML 文件的代码参见代码 13-4。

代码 13-4, index.ios.js 或者 index.android.js:

```
'use strict';
var React = require('react-native');
var {
  AppRegistry, StyleSheet, View, WebView, StatusBar
} = React;
var Project20 = React.createClass({
  render: function() {
    return (
      <View style={[styles.container]}>
        <StatusBar hidden={true}/>
        <WebView automaticallyAdjustContentInsets={false}
          style={styles.webView}
          source={require('./A.html')}//载入指定文件，注意文件名的大小写
          javaScriptEnabled={true}
          domStorageEnabled={true}
          startInLoadingState={true}/>
      </View>
    );
  },
});
var styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: 'blue',
  },
  webView: {
    backgroundColor: 'white',
    height: 350,
  }
});
AppRegistry.registerComponent('Project20', () => Project20);
```

//网页浏览器组件样式定义

13.4 音视频媒体播放

目前 React Native 框架还没有提供用于音视频媒体播放的组件。但是通过使用 `WebView` 组件，开发者可以访问制作好的、包含音视频媒体的网页，从而实现在 `React Native` 代码中进行简单的音视频媒体播放。包含音视频媒体的网页，可以存放在网络服务器上，也可以是存放在项目目录下的本地网页和本地音视频媒体文件。

`WebView` 组件可以指定它的大小和位置，结合专门为应用制作的网页，可以让应用的使用者完全无法分辨是使用 `WebView` 组件在播放音视频媒体。

因为具体实现涉及网页制作，这里不再继续讨论，只是指出一个实现的简单方向。如果开发者需要实现更多高级的音视频播放能力，则可以考虑使用混合开发。

第 14 章

图片的遍历、存取与显示

React Native 框架提供了 CameraRoll API 供开发者实现对手机中保存的图片、视频文件进行遍历访问与操作。在本书截稿时，React Native 发布到了 0.19.0 版本，CameraRoll API 刚刚实现了对 Android 平台的支持，成为一个跨平台的 API。目前 CameraRoll API 还未稳定，官方文档只有非常简单的说明。本章介绍的内容有可能趁着 React Native 新版本的发布而出现新的调整。

14.1 React Native 开发中 iOS 平台链接库的使用

如果使用 React Native 开发的移动应用支持 React Native 框架提供的所有功能，那么编译出的应用软件会非常大，并且其中可能会有些应用并没有使用到的功能。为了避免这种情况，React Native 以链接库的方式提供一些功能，当开发者使用到这些功能时，需要将链接库加入到项目中。CameraRoll API 是本书中第一个需要加入项目中的链接库。

提示：Android 平台不需要使用链接库，本节讨论的内容与 Android 平台毫无关系，但本章内容是跨平台实现的。

首先，开发者需要打开项目目录下的“\node_modules\react-native\Libraries\CameraRoll”文件夹，将这个文件夹中的 Xcode 项目文件 RCTCameraRoll.xcodeproj 拖动到 Xcode 项目的 Libraries 目录下，如图 14-1 所示。

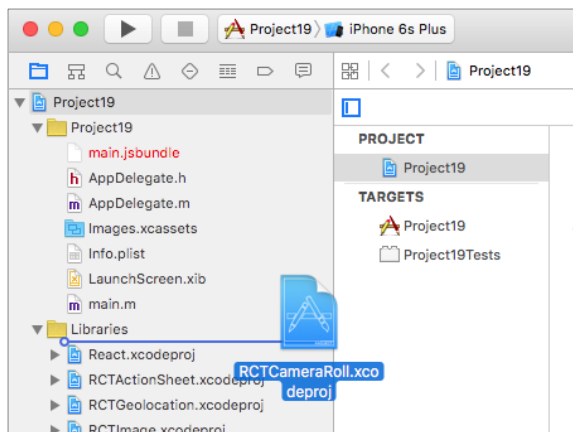


图 14-1 添加 RCTCameraRoll.Xcodeproj

添加成功后，项目文件列表如图 14-2 所示。

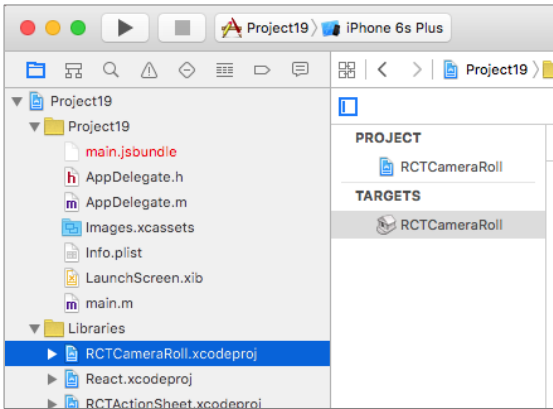


图 14-2 添加成功后的项目文件列表

然后，开发者需要选中上方 Project19 项目，在右侧的信息栏中选择“Build Phases”，点击打开“Link Binary With Libraries”子项。接下来点击打开刚才插入的 RCTCameraRoll.xcodeproj，再打开它的子目录 Products，将子目录下的 libRCTCameraRoll.a 文件拖动到“Link Binary With Libraries”子项列表中，如图 14-3 所示。

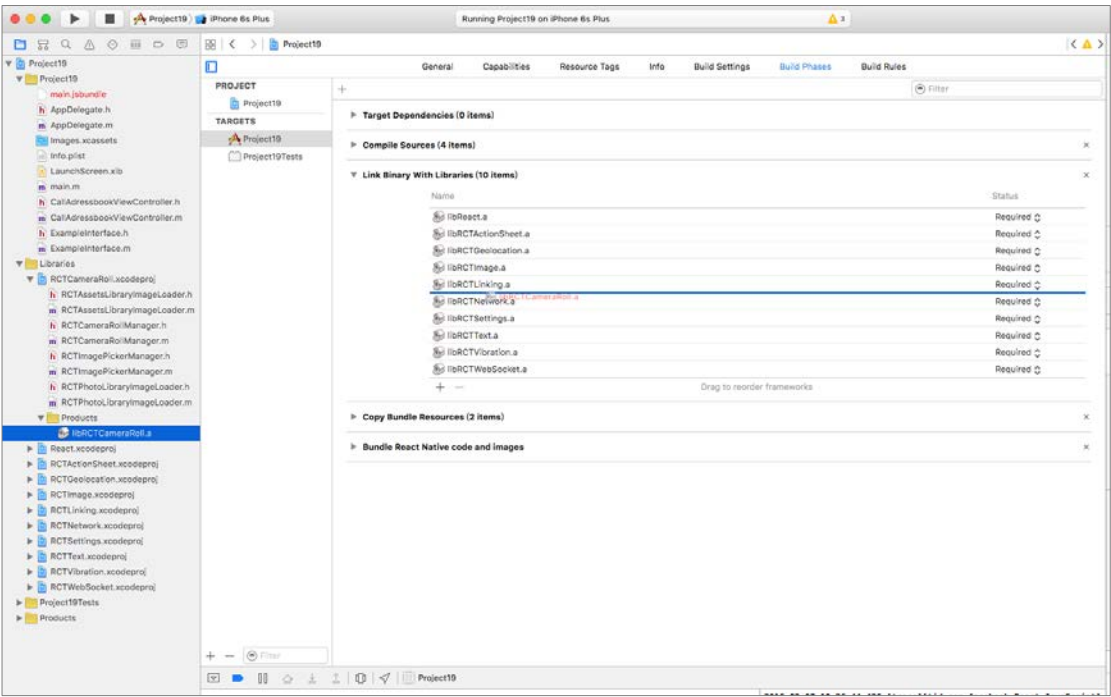


图 14-3 添加 libRCTCameraRoll.a

添加成功后，“Link Binary With Libraries”子项列表如图 14-4 所示。

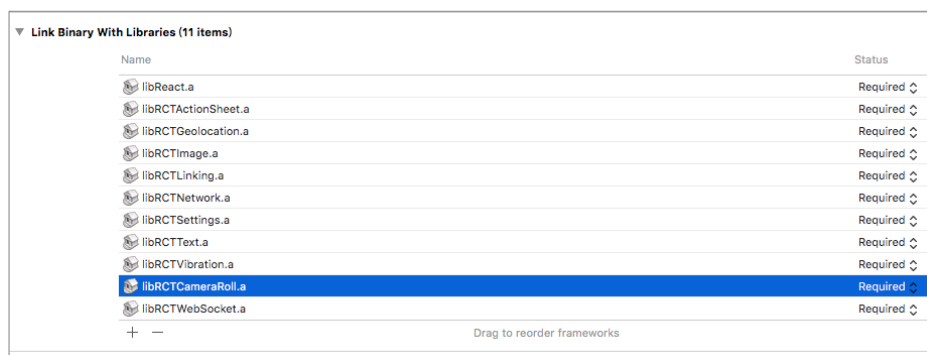


图 14-4 添加成功后的列表

14.2 获取手机中所有的图片信息

当移动应用需要对手机内图片、视频进行操作时，首先要考虑的就是手机中有多少张图片。这个数字会比较惊人，因为手机中的图片不仅有使用摄像头拍摄的照片、视频，还有各个应用从网络侧下载到手机中的图片与视频。

通常开发者需要做的第一步就是将符合某种规则(比如只取手机拍照的图片)的图片列出来，供用户在其中进行选择。

CameraRoll API 为开发者提供了 `getPhotos` 函数来实现这个功能。按官方文档以及开源代码注释中的说法，这个函数应该返回一个 `Promise` 对象供开发者进行下一步操作。但在 React Native 0.19.0 版本中，还没有实现。这个函数在 React Native 0.19.0 中的原型是：

```
static getPhotos(params: object, getResults: function, catchError:function)
```

在 React Native 0.20.0 版本中，同时支持回调函数和 `Promise` 机制。代码 14-3 将示范 `Promise` 机制的使用方法。

首先讨论 `getPhotos` 函数的第一个参数。第一个参数是一个对象，它可以有 4 个成员变量，分别是：

- `first`，数值型的成员变量，它必须要有，用来告诉 CameraRoll API 开发者希望获取多少张图片的信息。
- `groupTypes`，字符串类型的成员变量，默认值是 `SavedPhotos`。它还可以取值为：`Album`、`All`、`Event`、`Faces`、`Library`、`PhotoStream`。这个成员变量只在 iOS 平台上有效，用来指定获取图片或者视频的类型。
- `assetType`，字符串类型的成员变量，默认值是 `Photos`，表示只获取图片。它还可以取值为：`All`、`Videos`。
- `after`，字符串类型的成员变量，它的值用来记录上一次获取图片的截止标记。当参数中有它时，表示开发者希望从上一次截止的位置继续向下获取图片。它的值不能由开发者随意赋予，而是应当在上一次获取图片时得到并保存，以供下一次获取图片时使用。在 Android 平台上，开发者可以在一开始就提供这个成员变量，并在初始时将它赋值为 `null`；

但在 iOS 平台上，将它赋值为 null 会导致抛出一个无法捕捉的异常，进而导致应用出现大红屏界面。

getPhotos 函数的第二个和第三个参数都是回调函数，分别在成功和出错时回调。

使用 CameraRoll API 的例程片段如代码 14-1 所示。

代码 14-1:

```
.....
var {
  AppRegistry, StyleSheet, Text,
  View, CameraRoll, Image                //导入 CameraRoll
} = React;
var Project19 = React.createClass({
  fetchParams:{first:50,after:null},      //定义一个成员变量为获取图片时的参数
.....
componentWillMount:function() {          //开始获取图片
  CameraRoll.getPhotos(this.fetchParams, this.getPhotosResults, this.logError);
},
logError:function(error) {
  console.log("error:" + error);
},
getPhotosResults: function(data: Object) {
  var assets = data.edges;//将获得的图片文件数据都保存在 data.edges 中
  let len = assets.length;
  let asset;
  for( let i=0; i<len; i++) {
    asset=assets[i].node;
    .....//asset 中保存着一个图片文件的各项数据，对其进行处理
  }
  if (!data.page_info.has_next_page) {    //图片已经全部获取完
    return;
  }
  this.fetchParams.after = data.page_info.end_cursor;//更改获取图片参数中的 after
  //继续获取下 50 个图片文件
  CameraRoll.getPhotos(this.fetchParams, this.getPhotosResults, this.logError);
},
.....
```

当调用 getPhotos 函数获取图片成功时，回调函数会得到一个对象。这个对象是一个 JSON 对象，它的结构如下：

```
{
  edges: [
    node: {
      type: ...,
      group_name: ...,
      image: {
        uri: ...,
        height: ...,
        width: ...,
        isStored: ...,
      },
      timestamp: ...,
```

```

        location {
          latitude: ...,
          longitude: ...,
          altitude: ...,
          heading: ...,
          speed: ...,
        },
      },
      node: { ... },
      node: { ... },
      ...
    ],
    page_info: {
      has_next_page: ...,
      start_cursor: ...,
      end_cursor: ...,
    }
  }
}

```

14.3 图片信息详解

在代码 14-1 中，对于获取到的每一个图片文件，开发者都可以得到一个 `asset` 变量，变量中记录了图片文件的各项数据。现在我们来详细讨论这个变量中包含的各项数据及其用法。`asset` 同样是一个对象，在 Android 平台和 iOS 平台上，它有不同的成员变量。

14.3.1 Android 平台图片信息

在 Android 平台上，图片信息对象有如下成员：

- `type`，在 Android 平台上，它的值固定是“jpeg/jpg”，对开发者没有什么用处。
- `group_map`，字符串类型的变量，在 Android 平台上，它是保存当前图片文件的文件夹名。
- `timestamp`，数值型的变量，这是一个时间戳数值。
- `image`，它是一个对象，也是开发者最需要得到的数据。可以通过 `image.height` 得到图片的高度（数值型），通过 `image.width` 得到图片的宽度（数值型），通过 `image.uri` 得到一个内部的唯一标识。但它最重要的作用是，它作为一个整体，可以传递给 `Image` 组件，用来显示图片。

14.3.2 iOS 平台图片信息

在 iOS 平台上，图片信息对象有如下成员：

- `type`，字符串类型的变量，React Native 0.19.0 版本只有一个值：`ALAssetTypePhoto`。
- `group_map`，字符串类型的变量，React Native 0.19.0 版本只有一个值：`Camera Roll`。
- `timestamp`，数值型的变量，这是一个时间戳数值。
- `image`，它是一个对象，也是开发者最需要得到的数据。可以通过 `image.height` 得到图片的高度（数值型），通过 `image.width` 得到图片的宽度（数值型），通过 `image.uri` 得到

一个内部的唯一标识。但它最重要的作用是，它作为一个整体，可以传递给 Image 组件，用来显示图片。

- isStored，布尔类型的变量，对于获取到的图片，这个值都是 true。
- location，对象型的变量，有 latitude、longitude、altitude、heading、speed 五个成员变量。在 React Native 0.19.0 版本中，这些值全部是 undefined。从理论上说，一个相片文件可以在拍摄时携带拍摄地点的经纬度和海拔高度值。这个对象型的变量应当是预留的，尚未实现。

14.4 显示从 CameraRoll API 得到的图片

代码 14-2 通过 CameraRoll 获取一张图片，并且将图片显示在屏幕上。

代码 14-2，index.android.js:

```
'use strict';
var React = require('react-native');
var {
  AppRegistry, StyleSheet, View,
  CameraRoll, Image
} = React;
var Project19 = React.createClass({
  fetchParams:{first:1,after:null},
  getInitialState:function() {
    return {
      image:null
    };
  },
  componentWillMount:function() {
    CameraRoll.getPhotos(this.fetchParams, this.getPhotosResults, this.logError);
  },
  logError:function(error) {
    console.log("error:" + error);
  },
  getPhotosResults: function(data: Object) {
    let image = data.edges[0].node.image; //将获取的第一个图片文件中的 image 对象
    this.setState({image}); //放入状态机变量中
  },
  render: function() {
    if ( this.state.image === null ) return null; //尚未获取到图片文件信息
    return (
      <View style={styles.container}>
        <Image style={styles.imageStyle}
          source={this.state.image}/> //通过 CameraRoll 获取的图片信息被传入
      </View>
    );
  }
});
var styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
```

```

    alignItems: 'center',
    backgroundColor: '#F5FCFF',
  },
  imageStyle: {
    width: 300,
    height: 500
  }
});
AppRegistry.registerComponent('Project19', () => Project19);

```

代码 14-2 简单，运行效果不再给出。

14.5 为用户提供图片选择界面

现在，开发者可以获得手机上每一张图片的数据，也可以显示任意一张图片。通常开发者需要做的就是设计一个图片缩略图的列表，供用户在其中选择一张或者几张自己喜欢的图片。

开发选择界面的技术已经在第 8 章详细讨论过，在此不再重述。需要指出的是，结合 `getPhotos` 可以指定一次获取多少张图片信息，然后利用多次获取的特性，开发者可以很容易地实现第一次先呈现一定数量的图片，当划动到列表底部时，再继续获取的流程。`getPhotos` 可以优先返回最近拍摄的照片，因为通常用户在选取图片时都是希望选取最近拍摄的（或者下载保存的）图片，利用好这个特性，可以做到大概率让用户尽快选到希望选择的图片。

代码 14-3 示范了获取手机中的 30 张图片，并且以缩略图的形式将它们排列在手机屏幕上。

代码 14-3，`index.android.js`：

```

const React = require('react-native');
const {
  StyleSheet, Text, View, ScrollView, Image, CameraRoll, AppRegistry
} = React;
const Project20 = React.createClass({
  getInitialState: function() {
    return {
      images: [],
    };
  },
  componentDidMount: function() {
    const fetchParams = {
      first: 25,
    };
    //CameraRoll.getPhotos(fetchParams, this.storeImages, this.logError); //回调函数的写法
    //从 React Native 0.20.0 版本开始支持返回 Promise 对象，开发者应当优先使用 Promise 机制
    CameraRoll.getPhotos(fetchParams).then( (data) => {
      const assets = data.edges;
      const images = assets.map((asset) => asset.node.image);
      this.setState({images});
    }).catch( (error) => {
      console.log(error);
    });
  },
  storeImages: function(data) {
    const assets = data.edges;
    const images = assets.map((asset) => asset.node.image);
  },
});
AppRegistry.registerComponent('Project20', () => Project20);

```

```

    this.setState({images});
  },
  logError:function(error) {
    console.log(error);
  },
  render:function() {
    return (
      <ScrollView style={styles.container}>
        <View style={styles.imageGrid}>
          { this.state.images.map( (image) =>
            <Image style={styles.image}
              source={{ uri: image.uri }}
              key={ image.uri }/> ) //如果没有 key 这个属性,则会产生警告
          }
        </View>
      </ScrollView>
    );
  }
});
const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#F5FCFF',
  },
  imageGrid: {
    flex: 1,
    flexDirection: 'row',
    flexWrap: 'wrap',
    justifyContent: 'center'
  },
  image: {
    width: 100,
    height: 100,
    margin: 10,
  },
});
AppRegistry.registerComponent('Project20', () => Project20);

```

代码 14-3 在 Android 手机上(也可以在 iPhone 手机上运行)运行的效果如图 14-5 所示,具体图片依据各人手机上图片的不同而有所不同。

代码 14-3 只是图片选择界面实现的雏形。如果要在它的基础上继续实现,则需要读者自行加入以下功能:

- 用可触摸组件包含每张图片。
- 当 ScrollView 滑动到底部时,检测该事件并继续读取下 30 张图片,将新读取的图片加入显示中;

图片选择界面最好使用混合开发技术来实现,因此这里不再多述。具体理由请看 14.6 节。



图 14-5 代码 14-3 运行效果

14.6 图片的保存与读取显示

CameraRoll API 提供了 `saveImageWithTag` 方法。目前这个方法正处于调研开发阶段，还有待完善。

在移动应用开发中，开发者经常会遇到保存图片的功能需求。需求产生的原因主要有两个：

- 将来自网络 URI 的图片存储在手机本地，以备再次需要时直接从本地读取图片数据并显示。这种做法可以节省流量，并减轻服务器的数据读取压力。这种方法可能会产生在本地保存的图片与网络侧图片不一致的问题，需要通过客户端和服务端共同实现某种同步机制来解决。
- 保存手机存储中的某张图片，以防止这张图片被删除而无法再次获得。为了满足这个需求，需要保存的图片不能再以图片文件的格式保存在手机存储中，否则它还是会被手机的相册显示，从而有可能被删除。

React Native 框架的 `Image` 组件支持直接读入并显示以 `Base64` 格式编码的图片。利用这个能力，可以满足上述的功能需求。

在保存时，开发者需要做到：

- 获取图片数据，并且在需要时压缩图片以减少存储、显示图片时所需的磁盘空间和内存空间。
- 将压缩后的图片数据以 `Base64` 格式编码。
- 保存编码后的数据。

而在需要读取保存的图片时，开发者需要做到：

- 读取保存的图片。
- 显示图片。

14.6.1 保存图片数据

对需要保存的图片数据分为两种情况：

- 需要保存的图片数据是网络服务器侧准备好的。这意味着图片的宽和高都不需要客户端再次调整。但目前 React Native 没有提供相关能力供开发者直接得到图片文件中的数据，开发者需要使用混合开发的方式，通过原生代码的相关能力得到图片文件中的数据。
- 需要保存的图片是手机相机拍摄的，或者不是网络服务器侧为客户端准备的。这通常意味着在保存图片之前，开发者需要先压缩图片，缩减图片的宽和高。这个功能目前 React Native 代码无法实现，但使用原生代码可以很轻易地做到。因此开发者需要使用混合开发的方式来实现这个功能。在 React Native 代码的控制下，使用原生代码获取图片并对图片进行压缩，将压缩得到的数据再交给 React Native 侧代码。

得到图片数据后，接下来的工作是将数据用 `Base64` 格式编码。这一步也最好通过原生代码来实现，各平台都提供有相关能力。

最后一步是对原生代码传递给 React Native 侧代码的、以 Base64 格式编码的图片数据进行保存。它是一个字符串，这个字符串视获取时图片文件的格式不同而不同。它的内容是：

```
"data:image/png;base64,"+Base64 格式编码
```

或者

```
"data:image/jpeg;base64,"+Base64 格式编码
```

或者：

```
"data:image/gif;base64,"+Base64 格式编码
```

开发者可以把这个字符串按本书 7.3 节中讨论的技术保存在本地存储中。

14.6.2 读取并显示图片

当开发者使用 14.6.1 节中讨论的技术将图片文件保存在本地存储中后，接下来的需求就是在需要时读取并显示被保存的图片。为此，开发者需要：

- 按本书 7.3 节讨论的技术，通过 14.6.1 节中存储图片数据时使用的键值，将存储的数据取出并保存在一个字符串变量中。
- 显示这张图片。假设取出存储的数据并保存在名为 `imageDataReadFormAsyncStorage` 的字符串类型的状态机变量中，显示方法请读者参考代码 14-4；

代码 14-4：

```
.....
<Image source={{uri:this.state.imageDataReadFormAsyncStorage}}
      style={.....} //需要设置 width 与 height 样式键
.....
```

读者可以从 <https://github.com/xitaoque/14.5.2/tree/master> 中下载 `index.android.js` 文件（可以更名为 `index.ios.js` 在 iOS 平台上运行）。这段代码示范了如何在手机上显示一幅 Base64 编码的图片。因为 Base64 编码的图片数据长度比较长，故代码不在书中列出。

第 15 章

选择器、位置相关和应用状态

通常在移动应用中有两类选择器：一类是让用户选择一个日期或者选择一个时间；一类是提供有限多个值，让用户在这些值中选择一个。

本章介绍的两个组件和三个 API 都是在 React Native 0.19.0 版本中完成了跨平台实现的。目前相关文档非常少，同时功能还需要进行优化。相信当读者看到此书时，相应的完善工作都已经完成。

React Native 框架给开发者提供了 MapView 组件用来显示地图数据。在 React Native 0.19.0 版本中，它实现了对 Android 平台的支持。

React Native 框架给开发者提供了 AppState API 用来获取应用程序的当前状态。

15.1 日期、时间选择器

在移动应用开发中，提供相关界面让用户方便、快捷地选择日期和时间是经常遇到的功能需求。Android 平台和 iOS 平台对此都有各自的选择界面，开发者通过 React Native 开发可以很方便地调用各平台的选择界面满足移动应用的功能需求。

15.1.1 DatePickerAndroid API

在 Android 平台上，日期选择器是以 API，而不是以组件的形式提供的。日期选择器的调用、呈现方式与本书 3.9 节中介绍的 Alert API 类似。它的使用参见代码 15-1。

代码 15-1, index.android.js:

```
'use strict';
var React = require('react-native');
var {
  DatePickerAndroid, AppRegistry
} = React;
var Project20 = React.createClass({
  componentWillMount:function(){
    let today = new Date(); //创建一个 Date 对象，它记录了当前的日期和时间
    let theMinDate = new Date(2012,1,28); //创建一个 Date 对象，日期为 2012.1.28
    let theMaxDate = new Date( 2020,1,28); //创建一个 Date 对象，日期为 2020.1.28
    //组建复合对象以设置日期选择器的默认日期，最大与最小可选择日期
```

```

let option = { date:today, minDate:theMinDate, maxDate:theMaxDate};
//弹出一个日期选择窗口，窗口的默认日期是 today，向前最多能选择到 theMinDate，
//向后最多能选择到 theMaxDate
DatePickerAndroid.open(option).then( (result)=> {
  if ( result.action === DatePickerAndroid.dismissedAction) {
    console.log('用户没有选择日期');
    //用户没有选择日期，而是按下了返回键（取消按钮）关闭了选择日期窗口
  } else {
    console.log('用户选择了'+result.year+'年');
    console.log('用户选择了'+(result.month+1)+'月');
    console.log('用户选择了'+result.day+'日');
  }
}).catch( (error)=>{
  console.log('Error:'+error);
});
},
render: function() {
  return null;
}
});
AppRegistry.registerComponent('Project20', () => Project20);

```

在使用 `DatePickerAndroid` API 的 `Open` 函数时，可以传给它一个对象，用来设置默认日期、最小日期和最大日期。这三个值可以都不提供，不提供时，默认日期就是手机的当前日期，而最小日期、最大日期没有限制。这三个值也可以视用户需要提供其中任意一个。

当 Android 手机操作系统低于 5.0 时，设置最小日期和最大日期限制会导致 API 异常，最好不要设置，而是在用户选择完成后进行检查。

代码 15-1 被执行时，React Native 应用的原来界面会变暗，而图 15-1 所示的窗口将会被弹出显示在手机屏幕的正中央，用户可以在这个窗口中上下划动选择日期（可以通过触摸选择年、月、日）。



图 15-1 选择日期界面

开发者需要注意的是：DatePickerAndroid API 的 Open 函数调出来的界面不是 React Native 实现的界面，而是手机操作系统的界面。因此其调用简单，并且不能设置它的宽、高、字体等任何显示特性。这个界面通常是 Android 操作系统实现的，但在某些深度定制的手机上，手机厂商可能会修改这个界面，修改后的界面还是能完成同样的功能，只是在显示上会有些差异。

15.1.2 TimePickerAndroid API

使用 TimePickerAndroid API 的 Open 函数时，需要给它提供一个复合对象参数。它有三个成员变量，即 hour、minute 和 is24Hour，其中 hour 和 minute 是数值型，is24Hour 是布尔型。is24Hour 用来设置选择界面的小时值是使用 0~11 格式还是 0~23 格式，默认是 0~23 格式。虽然开发者可以设置 is24Hour，但在某些手机上这个参数不会产生作用。

TimePickerAndroid API 的用法请参见代码 15-2。

代码 15-2，index.android.js:

```
'use strict';
var React = require('react-native');
var {
  TimePickerAndroid, AppRegistry
} = React;
var Project20 = React.createClass({
  componentWillMount:function(){
    let theHour = 13; //设定时间选择器的默认小时值（0~23）
    let theMinute = 11; //设定时间选择器的默认分钟值（0~59）
    let aTime = { hour: theHour, minute: theMinute }; //组建复合对象
    //弹出一个时间选择窗口
    TimePickerAndroid.open( aTime ).then( (result)=> {
      if ( result.action !== TimePickerAndroid.timeSetAction) {

        console.log('用户没有选择时间');
        //用户没有选择时间，而是按下了返回键（取消按钮）关闭了选择时间窗口
      } else {
        console.log('用户选择了'+result.hour+'小时');
        console.log('用户选择了'+result.minute+'分钟');
      }
    }).catch( (error)=>{
      console.log('Error:'+error);
    });
  },
  render: function() {
    return null;
  }
});
AppRegistry.registerComponent('Project20', () => Project20);
```

在使用 TimePickerAndroid API 的 Open 函数时，可以不传给它任何参数，此时将默认使用手机当前时间。

当代码 15-2 被执行时，React Native 应用的原来界面会变暗，而图 15-2 所示的窗口将会被弹出显示在手机屏幕的正中央，用户可以在这个窗口中选择时间。

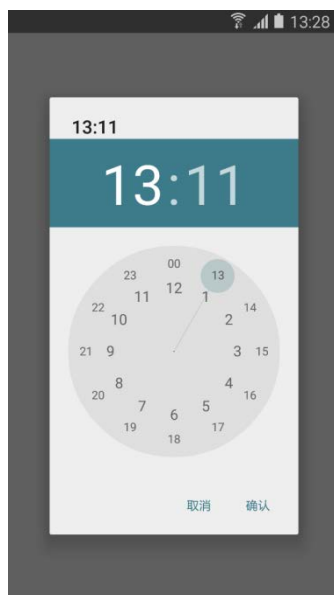


图 15-2 选择时间界面

同样，TimePickerAndroid API 的 Open 函数调出来的界面不是 React Native 实现的界面，而是手机操作系统的界面开发者无法控制任何显示特性。这个界面通常是 Android 操作系统实现的，但某些深度定制的手机可能会修改这个界面，修改后的界面还是能完成同样的功能，只是在显示上会有些差异。

15.1.3 DatePickerIOS 组件

React Native 为开发者提供了 DatePickerIOS 组件用来在 iOS 平台上选择日期、时间。相应功能在 Android 平台上是以 API 方式提供的，而在 iOS 平台上，则是以组件方式提供的，开发者可以将该组件与其他组件以各种方式排列在手机屏幕上。

DatePickerIOS 支持 View 组件的所有属性，这意味着开发者可以设置它的宽、高和位置等。

除了 View 组件的属性，DatePickerIOS 组件还支持下列属性：

- `date`，Date 类型，用来设置日期选择器的当前日期和时间；
- `maximumDate`，Date 类型，用来设置允许选择的最大日期；
- `minimumDate`，Date 类型，用来设置允许选择的最小日期；
- `minuteInterval`，数值类型，可取值有 1, 2, 3, 4, 5, 6, 10, 12, 15, 20, 30，用来设置可选的最小分钟单位；
- `mode`，字符串类型，可取值有 `date`、`time` 和 `datetime`，用来设置是选择日期、时间还是两者都选择；
- `onDateChange`，回调函数类型，当用户修改日期或时间时调用此回调函数。回调函数将接收一个 Date 类型参数，其中记录的是用户选择的日期和时间；

- `timeZoneOffsetInMinutes`，数值类型，以分钟为单位的时区时间差。在默认情况下，选择器会选择设备的默认时区。通过此参数，可以指定一个时区。例如，要使用北京时间（东八区），可以传递 $8 * 60$ 。

`DatePickerIOS` 组件的使用示例请参见代码 15-3。开发者需要注意的是，必须要把一个日期类型的状态机变量赋值给 `DatePickerIOS` 组件的 `date` 属性，并且在用户操作 `DatePickerIOS` 组件修改时间后，用新修改的时间值去更新相应的状态机变量；否则会出现用户使用 `DatePickerIOS` 组件修改了时间，几秒钟后，`DatePickerIOS` 组件又回到了原来的时间的情况。

代码 15-3:

```
'use strict';
var React = require('react-native');
var {
  AppRegistry, DatePickerIOS, Text, TextInput, View,
} = React;
var Project19 = React.createClass({
  getInitialState: function() {
    return {
      date1: new Date(),
      date2: new Date(),
      date3: new Date()
    };
  },
  timeZoneOffsetInHours: (-1) * (new Date()).getTimezoneOffset() / 60,
  onChange1: function(date1) {
    console.log("event1"+date1);
    this.setState({date1});
  },
  onChange2: function(date2) {
    console.log("event2"+date2);
    this.setState({date2});
  },
  onChange3: function(date3) {
    console.log("event3"+date3);
    this.setState({date3});
  },
  render: function() {
    return (
      <View>
        <DatePickerIOS date={this.state.date1}
          mode="datetime"
          timeZoneOffsetInMinutes={this.timeZoneOffsetInHours * 60}
          onChange={this.onChange1} />
        <DatePickerIOS date={this.state.date2}
          mode="date"
          timeZoneOffsetInMinutes={this.timeZoneOffsetInHours * 60}
          onChange={this.onChange2} />
        <DatePickerIOS date={this.state.date3}
          mode="time"
          timeZoneOffsetInMinutes={this.timeZoneOffsetInHours * 60}
          onChange={this.onChange3} />
      </View>
    );
  }
});
AppRegistry.registerComponent('Project19', () => Project19);
```

```

    );
  },
});
AppRegistry.registerComponent('Project19', () => Project19);

```

开发者需要注意的是，在模拟器上调试这个组件时，组件中月份、星期几的文字提示都是英文显示的，但在设置语言为中文的真机上运行时，文字提示将是中文的，如图 15-3 所示。



图 15-3 DatePickerIOS 组件 UI 效果

15.2 Picker 组件

Picker 组件可以在屏幕上弹出一个窗口，供用户在所提供的选项中选择一个。

15.2.1 Picker 组件的样式设置

Picker 组件支持 View 组件的所有属性，也就包含了 View 组件的样式设置。此外，Picker 组件的样式设置还额外支持 color 键。

15.2.2 Picker 组件的属性

onValueChange 是回调函数属性，当用户选择 Picker 的某一选项时，回调函数将被执行并带有如下参数：

- itemValue，被选中项的 value 属性；
- itemPosition，被选中项在 Picker 中的索引位置。

selectedValue 属性可以是字符串类型或者数值类型，用来指定默认选中的值。

15.2.2.1 Android 平台特有属性

enabled 是布尔类型的属性。如果它的值为 false，则本 Picker 组件不可选择。

`mode` 是字符串类型的属性。它可以取值 `dialog` 或者 `dropdown`, 用来定义 `Picker` 的呈现方式。当它取值为 `dialog` 时, 将在手机屏幕中部弹出一个选择框, 列出选项让用户选择; 当它取值为 `dropdown` 时, 将在 `Picker` 组件的当前位置弹出一个下拉框, 列出选项让用户选择。在 `Android` 平台上, 此属性的默认值是 `dialog`。

`prompt` 是字符串类型的属性。当 `mode` 为 `dialog` 时, 这个字符串将会呈现在选择框的首行。

15.2.2.2 iOS 平台特有属性

`itemStyle` 是样式属性, 用来指定应用在每个选项标签上的样式。这个样式可以按照 `Text` 组件的样式进行设置。

15.2.3 Picker.Item 组件属性

`Picker.Item` 组件用来声明一个 `Picker` 组件所拥有的选项。它有三个属性:

- `label`, 字符串类型属性, 用来指定选项的显示字符串;
- `color`, 颜色类型属性, 用来指定选项的显示颜色;
- `value`, 字符串类型属性, 用来指定选项的返回值, 当选项被选中时, 这个值被 `onValueChange` 回调函数传回。

15.2.4 Picker 组件例程

代码 15-4 示范了 `Picker` 组件的使用。

代码 15-4, `index.android.js`:

```
'use strict';
const React = require('react-native');
const {
  Picker, AppRegistry, StyleSheet, Text, View,
} = React;
const Item = Picker.Item;
var Project19 = React.createClass({
  options: ['选项一', '选项二', '选项三', '选项四', '选项五'],
  getInitialState: function() {
    return {
      choice: ''
    };
  },
  render: function() {
    return (
      <View style={styles.container} >
        <Picker
          style={styles.picker}
          mode={Picker.MODE_DROPDOWN} //这一行去掉就是弹出窗口效果
          prompt='提示字符串'
          selectedValue={this.state.choice}
          onValueChange={this.onValueChange}>
          { this.options.map( (aOption) =>
```

```

        <Item label={aOption}
            value={aOption}
            key={aOption}/> )
    }
    </Picker>
    <Text style={styles.welcome}>
        {'\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n'}你选择了:{this.state.choice}
    </Text>
</View>
);
},
onValueChange: function(choice: string, noUse: string) {
    this.setState({choice});
},
});
var styles = StyleSheet.create({
    container: {
        flex: 1,
        justifyContent: 'center',
        alignItems: 'center',
        backgroundColor: '#F5FCFF',
    },
    welcome: {
        fontSize: 30,
        textAlign: 'center',
        margin: 10,
    },
    picker:{
        width:200,
        height:30
    }
});
AppRegistry.registerComponent('Project19', () => Project19);

```

代码 15-4 的运行效果如图 15-4 所示。这是应用刚启动时，用户还没有选择时的效果。如图 15-5 所示，左图是下拉框模式，右图是弹出框模式。

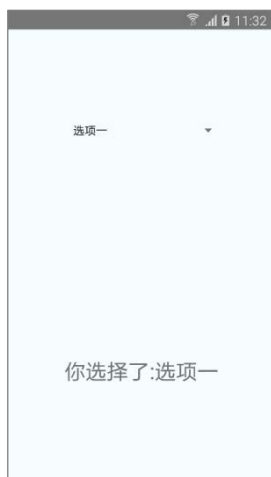


图 15-4 Android 平台 Picker 组件 UI 效果（一）



图 15-5 Android 平台 Picker 组件 UI 效果（二）

在 iOS 平台上, Picker 组件不再需要 mode 属性, 但需要提供 itemStyle 属性。在 iOS 平台上使用 Picker 组件的示例请参见代码 15-5。

代码 15-5, index.ios.js:

```
'use strict';
const React = require('react-native');
const {
  Picker, AppRegistry, StyleSheet, Text, View,
} = React;
const Item = Picker.Item;
var Project20 = React.createClass({
  options: ['选项一', '选项二', '选项三', '选项四', '选项五'],
  getInitialState: function() {
    return {
      choice: ''
    };
  },
  render: function() {
    console.log('picker start. ');
    if (Picker === null ) console.log('picker is null');
    if (Picker === undefined ) console.log('picker is undefined');
    return (
      <View style={styles.container} >
        <Picker
          style={styles.picker}
          selectedValue={this.state.choice}
          onValueChange={this.onValueChange}
          itemStyle={styles.pickerItemStyle}>
          { this.options.map( (aOption) =>
            <Item label={aOption}
              value={aOption}
              key={aOption}/> )
          }
        </Picker>
        <Text style={styles.welcome}>
```

```

        {'\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n'}你选择了:{this.state.choice}
      </Text>
    </View>
  );
},
onValueChange: function(choice: string, noUse: string) {
  this.setState({choice});
},
});
var styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#F5FCFF',
  },
  picker:{
    width:200,
    height:600
  },
  pickerItemStyle:{
    width:100,
    height:600
  }
});
AppRegistry.registerComponent('Project20', () => Project20);

```

代码 15-5 在 iOS 平台上运行的效果如图 15-6 所示。



图 15-6 iOS 平台 Picker 组件 UI 效果

接下来要讨论的 `PickerIOS` 组件的界面与此类似，但是它有一种滚筒的效果。在开发中具体使用哪个组件，由开发者视项目需要以及选项多少来综合考虑决定。

15.3 PickerIOS

PickerIOS 是专用于 iOS 平台的组件，并且它没有对应的 Android 平台组件。

PickerIOS 的用法与 Picker 非常相似，所以这里只介绍不同点。

- PickerIOS 组件不需要设置 style 属性；
- PickerIOS 组件的父组件不能设置 style 属性。

代码 15-6 示范了 PickerIOS 组件的使用。

代码 15-6, index.ios.js:

```
'use strict';
const React = require('react-native');
const {
  PickerIOS, AppRegistry, StyleSheet, Text, View,    //导入 PickerIOS 组件
} = React;
var PickerItemIOS = PickerIOS.Item;                  //导入 PickerItemIOS 组件
var Project19 = React.createClass({
  options:['选项一','选项二','选项三','选项四','选项五'],
  getInitialState: function() {
    return {
      choice:''
    };
  },
  render: function() {
    return (
      <View >          //这个View不能有 style 属性，否则显示异常
        <PickerIOS
          selectedValue={this.state.choice}
          onValueChange={this.onValueChange}>
          { this.options.map( (aOption) =>
            <PickerItemIOS label={aOption}
              value={aOption}
              key={aOption}/> )
          }
        </PickerIOS>
        <Text style={styles.welcome}>
          {'\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n'}你选择了:{this.state.choice}
        </Text>
      </View>
    );
  },
  onValueChange: function(choice: string, choice1: string) {
    this.setState({choice});
  },
});
var styles = StyleSheet.create({
  welcome: {
    fontSize: 30,
    textAlign: 'center',
    margin: 10,
  }
});
```



```
AppRegistry.registerComponent('Project19', () => Project19);
```

代码 15-6 运行效果如图 15-7 所示。



图 15-7 PickerIOS 组件 UI 效果

在 iOS 平台上，为了使风格与 iOS 应用相似，建议开发者还是多使用 PickerIOS 组件，而不要使用 Picker 组件。

15.4 MapView 组件

React Native 为开发者提供了 MapView 组件用来在手机 UI 上展示地图信息。开发者可以指定希望展示的地图范围和地图比例。

在 React Native 0.19.0 版本中，第一次宣布 MapView 组件将成为一个跨平台组件。MapView 组件还在发展演进之中，当读者看到这本书时，MapView 组件也许已经有了比较多的完善。关于更多的 MapView 组件的使用，请读者参阅官方文档。

从另一方面来说，虽然 MapView 组件正慢慢发展成一个跨平台组件，但从它的 API 可以明显看出来，iOS 平台上的 MapView 组件的能力要远远强于 Android 平台上的 MapView 组件。这会给开发者造成一些困扰：当实现一个跨平台应用时，一个普遍的要求是应用在两个平台上运行时应当基本具备相同的能力。在不同的平台上运行时，在 UI 上可以有些差异，但功能应当大致相同。要使用 MapView 组件实现地图相关功能，又要做到功能相同，就只能使用 MapView 组件在两个平台上都具备的能力。本节也仅限于介绍这些可以跨平台实现的能力。

如果地图相关功能是正在开发的的核心能力，那么开发者应当考虑通过混合开发来实现这个应用。与地图功能相关的 UI 和业务逻辑使用原生代码来实现，在不同的平台上各实现一套。

百度地图、高德地图都有开放的 API 供开发者在原生代码中使用。

15.4.1 MapView 组件样式设置

MapView 组件支持 View 组件的所有属性，其中就包括 View 的样式设置。MapView 组件没有自己特殊的样式设置。

15.4.2 MapView 组件特有的跨平台属性

`pitchEnabled` 是布尔类型的属性，用来设置是否允许用户通过两指触屏旋转这个操作改变地图的视角。它的默认值是 `true`（允许）。当它为 `false` 时，MapView 组件显示地图的视角永远是垂直的，用户查看地图的感觉始终是在地图的正上方垂直向下查看。

`rotateEnabled` 是布尔类型的属性，用来设置是否允许用户通过两指触屏旋转这个操作改变地图的方向。它的默认值是 `true`（允许）。当它为 `false` 时，MapView 组件显示的地图永远是“上北下南，左西右东”。

`region` 是一个对象类型的属性。它的数据结构是：

```
{
  latitude: number,
  longitude: number,
  latitudeDelta: number,
  longitudeDelta: number
}
```

它可以用来设置 MapView 显示的地图区域。其中，`latitude` 与 `longitude` 定义地图显示的中心点坐标；两个 `Delta` 变量通过定义经纬度的增减值来确定地图显示的范围。

`onRegionChange` 是回调函数属性。当开发者设置此属性时，用户拖拽地图时此回调函数将被持续调用并被传入一个 `region` 对象。

`onRegionChangeComplete` 是回调函数属性。当开发者设置此属性时，用户拖拽地图结束时此回调函数将被调用并被传入一个 `region` 对象。

`scrollEnabled` 是布尔类型的属性，默认值为 `true`，用来设置是否允许用户改变地图所显示的区域。

`showsUserLocation` 是布尔类型的属性，默认值为 `false`。当它为 `true` 时，MapView 会显示用户所在位置的地图（如果能获取到位置信息）；如果把它设置为 `true`，开发者还需要在 Xcode 工程的 `Info.plist` 中增加 `NSLocationWhenInUseUsageDescription` 字段，否则应用会没有任何异常提示直接退出。

为了在 Xcode 工程的 `Info.plist` 文件中增加 `NSLocationWhenInUseUsageDescription` 字段，需要点击项目列表中的 `Info.plist` 文件，然后在右侧的 Key 中找到 `NSLocationWhenInUseUsageDescription` 字段（如果没有则新建这个字段），将其对应的值修改为 YES，如图 15-8 所示。

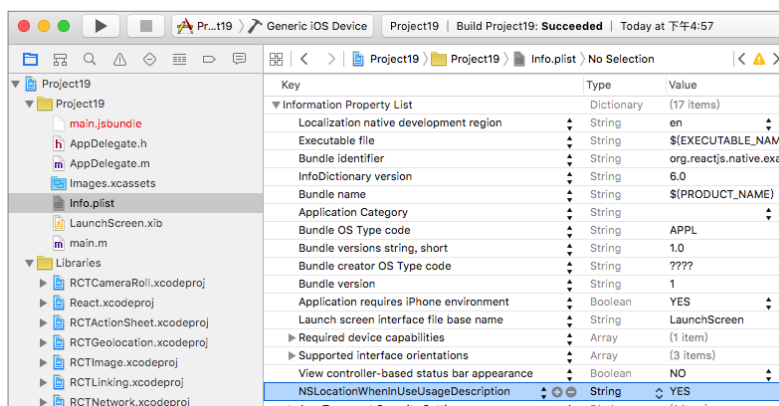


图 15-8 增加 NSLocationWhenInUseUsageDescription 字段

zoomEnabled 是布尔类型的属性，默认值为 true，用来设置是否允许用户旋转、缩放地图。

MapView 组件还有一些非跨平台属性，这些属性在本书中不予讨论。

onAnnotationPress 是回调函数属性。现在不鼓励使用它，而是鼓励使用一个 iOS 平台独有的属性来替代。它在用户点击 MapView 组件中的注释点时被调用，而 MapView 组件中的注释点只能在 iOS 平台上设置，因此虽然它被列为跨平台属性，但事实上是 iOS 独有的属性。

15.4.3 MapView 组件例程

代码 15-7 使用 MapView 组件在手机界面上呈现出北京邮电大学及其附近区域的地图。

代码 15-7, index.ios.js:

```
'use strict';
const React = require('react-native');
const {
  AppRegistry, MapView, PropTypes, StyleSheet, Text, TextInput, View
} = React;
var Project19 = React.createClass({
  buptRegion:{
    latitude: 39.961737,           //北京邮电大学校区的纬度
    longitude: 116.357655,        //北京邮电大学校区的经度
    latitudeDelta: 0.008,         //显示区域大小
    longitudeDelta: 0.008        //显示区域大小
  },
  getInitialState() {
    let Dimensions = require('Dimensions');
    let totalWidth = Dimensions.get('window').width;
    let totalHeight = Dimensions.get('window').height;
    return {
      mapStyle:{
        width:totalWidth,
        height:totalHeight
      },
      region: this.buptRegion
    };
  }
});
```

```

    },
    _onRegionChange(region) {                                //不对这个事件进行任何处理
    },
    _onRegionChangeComplete(region) {                          //当用户拖移地图结束时，在日志中打印出
        console.log( region.latitude);                        //当前显示区域的各项参数
        console.log( region.longitude);
        console.log( region.latitudeDelta);
        console.log( region.longitudeDelta);
    },
    _onLayout:function(event) {                                //当屏幕显示方向改变时，改变地图的显示参数依然以整屏显示
        let {width,height} = event.nativeEvent.layout;
        if ( width === this.state.mapStyle.height ) {
            let mapStyle = {};
            mapStyle.width = this.state.mapStyle.height;
            mapStyle.height = this.state.mapStyle.width;
            this.setState({mapStyle});
        }
    },
    render: function() {
        return (
            <View onLayout={this._onLayout}>
                <MapView
                    style={this.state.mapStyle}
                    region={this.state.region}
                    onRegionChange={this._onRegionChange}
                    onRegionChangeComplete={this._onRegionChangeComplete}/>
            </View>
        );
    }
  });
AppRegistry.registerComponent('Project19', () => Project19);

```

代码 15-7 运行效果如图 15-9 所示。



图 15-9 代码 15-7 运行效果

在代码 15-7 中，如果希望显示用户的位置，则需要给 MapView 加入属性：

```
showsUserLocation={true}
```

开发者要注意，如果仅是加入这个属性，那么只有当前的用户位置在 MapView 显示的地图范围内时，才会在地图上有个蓝色的小圆圈用来标记用户的可能位置；否则 MapView 显示的地图不会有任何变化。

在 iOS 平台上，MapView 组件还有一个 followUserLocation 属性，将它设置为 true 后，与 showsUserLocation 配合，可以直接定位并显示用户位置周边的地图。这时 MapView 组件的整个声明参见代码 15-8。

代码 15-8：

```
<MapView
  style={this.state.mapStyle}
  followUserLocation={true}
  showsUserLocation={true}
  onRegionChange={this._onRegionChange}
  onRegionChangeComplete={this._onRegionChangeComplete}/>
```

代码 15-8 可以很方便地显示用户位置周边的地图，但可惜的是，代码 15-8 只能运行在 iOS 平台上。

15.5 AppState API

通过 React Native 提供的 AppState API，开发者可以时刻得知应用当前在前台运行还是在后台运行。

15.5.1 AppState API 用途与用法

AppState API 只有两个对外的静态函数：addEventListener 和 removeEventListener。

使用方法是：当应用启动（或者需要监听应用状态）时，添加监听器监听应用状态；当应用结束（或者不再需要监听应用状态）时，释放监听器。

收到的应用状态字符串有三个预定义值：active、inactive 和 background。其中 inactive 是预留的，目前监听程序不会收到这个状态。

应用启动时，应用状态肯定是 active，启动监听器后，不会收到这个状态的提示。

15.5.2 AppState API 例程

代码 15-9 演示了 AppState API 的用法。

代码 15-9：

```
'use strict';
const React = require('react-native');
const {
```

```

    AppRegistry, AppState //导入 AppState API
  } = React;
  var Project19 = React.createClass({
    render: function() {
      return null; //因为手机屏幕不需要有任何显示，所以直接返回 null
    },
    componentWillMount: function() { //启动监听器
      AppState.addEventListener('change', this._handleAppStateChange);
    },
    componentWillUnmount: function() { //释放监听器
      AppState.removeEventListener('change', this._handleAppStateChange);
    },
    _handleAppStateChange: function(newState) {
      console.log( 'Appstate is:'+newState); //在日志中打印出监听到的新的应用状态
    }
  });
  AppRegistry.registerComponent('Project19', () => Project19);

```

在 Android 平台和 iOS 平台上，当用户按 Home 键切换出应用时，在日志中会打印出当前应用状态进入 background；而当用户再次切换入应用时，在日志中会打印出当前应用状态进入 active。日志输出如图 15-10 所示。

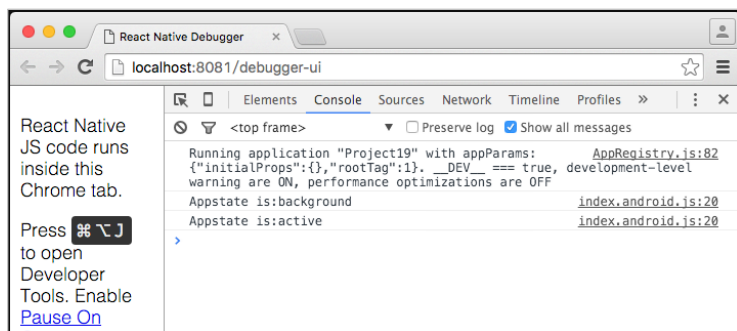


图 15-10 代码 15-9 日志输出

15.6 获取地理位置

从 React Native 0.19.0 版本开始，开发者可以通过全局变量 `navigator.geolocation` 的三个静态方法来进行获取地理位置相关操作。这三个函数分别是：`getCurrentPosition`、`watchPosition` 和 `clearWatch`。

如果要获取地理位置，则需要保证平台项目配置有对应的权限。

在 iOS 平台上，在项目的 `Info.plist` 文件中要有 `NSLocationWhenInUseUsageDescription` 键。

在 Android 平台上，在项目的 `AndroidManifest.xml` 文件中要有下列权限：

```
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
```

修改了 React Native 项目目录下的 `\android\app\src\main\ AndroidManifest.xml` 文件后，需要重新在项目目录下输入“`react-native run-android`”命令编译项目，并将编译后应用安装到手机上。

代码 15-10 示范了如何获取地理位置信息。

代码 15-10, index.ios.js 或者 index.android.js:

```
'use strict';
var React = require('react-native');
var {
  AppRegistry, StyleSheet
} = React;
var Project20 = React.createClass({
  watchID:null, //监听设备地理位置变化的 ID, 取消监听时要用到
  getPositionResult:function(aPosition) {
    console.log(aPosition); //处理得到的位置信息, 这里只是简单打印
  },
  logError:function(aError) {
    console.log(aError); //处理位置信息时发生的错误
  },
  componentDidMount: function() {
    let para = {
      enableHighAccuracy: true, //允许高精度定位
      timeout: 20000, //20 秒没得到位置信息停止获取
      maximumAge: 1000 //定位结果缓存 1000 毫秒
    };
    //下一条语句请求获取地理位置信息, 结果视情况异步交给两个回调函数中的一个
    navigator.geolocation.getCurrentPosition(
      this.getPositionResult,this.logError, para);
    //下一条语句启动地理位置变化监听器
    this.watchID = navigator.geolocation.watchPosition(this.getPositionResult);
  },
  componentWillUnmount: function() {
    //应用退出前, 关闭地理位置变化监听器
    navigator.geolocation.clearWatch(this.watchID);
  },
  render: function() {
    return null;
  }
});
AppRegistry.registerComponent('Project20', () => Project20);
```

成功获得地理位置信息时, 回调函数将收到一个包含地理位置信息的对象。这个对象的数据结构是:

```
{
  coords: {
    accuracy: 65, //数值类型, 精度
    altitude: 47.8149299621582
    altitudeAccuracy: 11.17654172089324
    heading: -1
    latitude: 49.9498382716754
    longitude: 116.3420437341329
    speed: -1
  }
  timestamp: 477688809904.147
}
```

开发者通过回调函数中传入的对象得到地理位置信息，按照业务流程进行处理。

15.7 VibrationIOS API

React Native 在 iOS 平台上为开发者提供了 VibrationIOS API 用来实现手机振动功能。它的使用方法非常简单。

```
var {  
  AppRegistry, StyleSheet, View, Text, VibrationIOS      //导入 VibrationIOS API  
} = React;  
  
.....  
    VibrationIOS.vibrate();          //在需要实现手机振动时调用 vibrate 函数  
.....
```

每调用一次 vibrate 函数，手机就会振动 1 秒钟时间。如果想实现按指定频率振动，则需要配合定时器来完成。

第 16 章

使用 ES 6 语法开发

使用 ES 5 语法编写的 React Native 代码，基本上可以按照一一对应的关系改为使用 ES 6 语法编写。

在阅读本章之前，读者需要知道，目前使用 ES 5 语法进行 React Native 开发并没有被不鼓励。使用 ES 6 语法还无法实现“mixins”这个在 React Native 开发中经常使用到的特性，所以有些代码还是使用 ES 5 语法编写方便。

从 ES 5 语法开发转入 ES 6 开发主要需要注意下面讨论的 6 个部分。

16.1 React Native 组件导入

使用 ES 5 语法时，我们使用代码 16-1 来导入 React Native 组件。

代码 16-1:

```
let React = require('react-native');
let {
  AppRegistry,
  StyleSheet,
  Text,
  View
} = React;
```

使用 ES 6 语法时，需要将代码 16-1 改为代码 16-2 的形式。

代码 16-2:

```
import React, {
  AppRegistry,
  Component,
  StyleSheet,
  Text,
  View
} from 'react-native';
```

注意，在代码 16-2 中，开发者需要多导入一个 Component 组件。

开发者使用 ES 5 语法创建自己的 React Native 组件时，代码片段如代码 16-3 所示。

代码 16-3:

```
let Project18 = React.createClass({
  .....,
}),
```

使用 ES 6 语法时，需要将代码 16-3 修改为代码 16-4 的形式。

代码 16-4:

```
class Project18 extends Component {
  .....
}
```

注意，代码 16-4 以反大括号结束，不需要 ES 5 最后结尾的反小括号和逗号。

16.2 属性声明

本书 3.7 节和 3.8 节中详细讨论了如何声明属性和指定属性默认值。

使用 ES 5 语法时，它们的写法如代码 16-5 所示。

代码 16-5:

```
.....
var Project18 = React.createClass({
  propTypes: {
    aStringProp: React.PropTypes.string
  },
  getDefaultProps: function() {
    return { aStringProp: 'default value' };
  }
  .....
});
.....
```

使用 ES 6 语法时，需要将代码 16-5 修改为代码 16-6 的形式。

代码 16-6:

```
.....
var Project18 extends Component {.....}
Project18.propTypes = {
  aStringProp: React.PropTypes.string
};
Project18.defaultProps = {
  aStringProp: 'default value'
};
```

注意：在 ES 6 语法中，属性的类型声明和默认值声明不像 ES 5 语法那样在组件定义内部声明，而是在组件定义外部声明。

16.3 成员变量声明

使用 ES 5 语法时，React Native 组件的成员变量声明如代码 16-7 所示。

代码 16-7:

```
.....
let Project18 = React.createClass({
  _myProperty1: 'test',
  _myProperty2: true,
  .....

```

使用 ES 6 语法编写 React Native 组件时，可以为 React Native 组件指定一个构造函数，而声明 React Native 组件的成员变量必须在组件的构造函数中声明。代码 16-7 需要修改为代码 16-8 的形式。

代码 16-8:

```
.....
var Project18 extends Component {
  constructor(props) {          //组件构造函数名称与声明方式，不可修改
    super(props);               //将属性传给父类构造函数，必须有这句，不可修改
    this._myProperty1='text';   //声明成员变量
    this._myProperty2=true;     //声明成员变量
  }
  .....

```

使用 ES 6 语法编写 React Native 组件时函数名称后不能再有“: function”关键字，不同于使用 ES 5 语法编写时可有可没有这个关键字。

16.4 状态机变量声明

使用 ES 5 语法时，React Native 组件的状态机变量声明如代码 16-9 所示。

代码 16-9:

```
.....
let Project18 = React.createClass({
  getInitialState:function() {
    return{
      var1: 'value of var1',
      var2:30,
      var3:true
    };
  }
  .....

```

使用 ES 6 语法编写 React Native 组件时，React Native 组件的状态机变量也必须在组件的构造函数中声明。代码 16-9 需要修改为代码 16-10 的形式。’

代码 16-10:

```
.....
var Project18 extends Component {
  constructor(props) {          //组件构造函数名称与声明方式，不可修改
    super(props);               //将属性传给父类构造函数，构造函数中必须有这句，不可修改
    this.state = {              //声明状态机变量
      var1: 'value of var1',
      var2:30,

```

```

        var3:true
    };
}
.....

```

16.5 回调函数绑定

使用 ES 5 语法时，React Native 组件的回调函数可以直接指向本组件的某个成员方法。示例如代码 16-11 所示。

代码 16-11:

```

.....
let Login = React.createClass({
  getInitialState:function() {
    return{
      inputedNum: ''
    };
  },
  updateNum: function(newText) {           //这个成员方法将被用作回调函数
    this.setState((state) => {
      return {
        inputedNum: newText
      };
    });
  },
  buttonPressed: function() {             //这个成员方法将被用作回调函数
    .....
  },
  render: function() {
    return (
      <View style={styles.container}>
        <TextInput style={styles.numberInputStyle}
          placeholder={'请输入手机号'}
          //下面的语句指定第一个回调函数
          onChangeText={(newText) => this.updateNum(newText)}/>
        <Text style={styles.bigTextPrompt}
          onPress={this.buttonPressed}>//指定第二个回调函数
          确定
        </Text>
      </View>
    );
  }
});
.....

```

使用 ES 6 语法编写 React Native 组件时，React Native 组件的回调函数必须在组件的构造函数中执行绑定操作。使用 ES 5 语法开发也是有这一步绑定操作的，但 React 类的 `creatClass` 方法为开发者代劳了这一步操作。使用 ES 6 语法开发这一步必须由开发者自己在代码中绑定。代码 16-11 需要修改为代码 16-12 的形式。

代码 16-12:

```

.....
class Login extends Component {
  constructor(props) {
    super(props);
    this.state = {
      inputNum: ''
    };
    //下面两条语句将两个回调函数和成员方法绑定
    this.updateNum = this.updateNum.bind(this);
    this.buttonPressed = this.buttonPressed.bind(this);
  }
  updateNum(newText) {
    this.setState((state) => {
      return {
        inputNum: newText
      };
    });
  }
  buttonPressed() {
    .....
  }
  render() {
    return (
      <View style={styles.container}>
        <TextInput style={styles.numberInputStyle}
          placeholder={'请输入手机号'}
          onChangeText={(newText) => this.updateNum(newText)} />
        <Text style={styles.bigTextPrompt}
          onPress={this.buttonPressed}>
          确定
        </Text>
      </View>
    );
  }
}
.....

```

相对于使用 ES 5 语法开发,使用 ES 6 语法开发需要开发者通过代码自己绑定每一个回调函数,这对开发者来说是一种开发方便性上的退步。但直到 React Native 可以使用 ES 7 语法开发前,只能使用这种不方便的办法。

16.6 类的静态成员变量与静态成员函数

使用 ES 5 语法进行 React Native 开发时,类的静态成员变量与静态成员函数的实现如代码 16-13。

代码 16-13:

```

.....
statics: {
  _myStaticObject: 'init value',           //定义类的静态成员变量
  myStaticMethod: function() {           //定义类的静态成员函数

```

```
        console.log('myStaticMethod is called.')
      }
    },
    render: function() {
      console.log(Project19._myStaticObject); //访问类的静态成员变量
      Project19.myStaticMethod();           //调用类的静态成员函数
    },
    .....
  },
  .....
}
```

使用 ES 6 语法开发时，相应的实现如代码 16-14。

代码 16-14:

```
class Project19 extends Component {
  static myStaticObject = 'init value'; //定义类的静态成员变量
  static myStaticMethod () {           //定义类的静态成员函数
    console.log('myStaticMethod is called.');
```

```
  }
  .....
  render() {
    console.log(Project19._myStaticObject); //访问类的静态成员变量
    Project19.myStaticMethod();           //调用类的静态成员函数
  },
  .....
}
```

第 17 章

混合开发高级篇

在高级混合开发中，将原生代码实现的 UI 模块或者其他功能封装成一个 React Native 组件，这种组件被称为私有的 React Native 组件。开发者可以在 React Native 代码中直接调用私有的 React Native 组件。如果两个平台各自使用原生代码实现了一个同名的私有的 React Native 组件，则可以很方便地实现跨平台移动应用开发。

17.1 使用 Objective-C 语言创建私有的 React Native 组件

将 Objective-C 实现的 UI 组件封装成 React Native 组件的相关基础技术，React Native 官方文档已经写得非常详细完备了，开发者在需要实现此功能时，一定要去阅读相关文档。英文文档的地址是 <http://facebook.github.io/react-native/docs/native-components-ios.html#content>，中文文档的地址是 <http://reactnative.cn/docs/native-component-ios.html#content>。官方文档详尽地描述了 iOS 平台的 MapView 组件是如何被封装与实现的。

本节将在官方文档的基础上，把 FLAnimatedImage 这个开源项目封装成一个私有的 React Native 组件，并通过 React Native 代码来调用这个组件。在例程实现的过程中，从另一个侧面讨论封装私有组件时需要注意的事项，React Native 框架为其对应的原生代码视图类提供的生命周期函数，以及如何利用这些生命周期函数。

FLAnimatedImage 开源项目可以在 iOS 平台上显示 GIF 动画。这个开源项目地址是：<https://github.com/Flipboard/FLAnimatedImage>。

17.1.1 增加 FLAnimatedImage 链接库

首先需要下载压缩包，下载后将它解压在项目目录的 iOS 子目录下，不更换名称的话，解压出来的文件夹应当是 FLAnimatedImage-master。

在 FLAnimatedImage-master 文件夹下有个名为 FLAnimatedImage.Xcodeproj 的 Xcode 项目文件，将这个项目文件拖入 React Native 项目的 Libraries 下，再将项目生成文件加入库中，如图 17-1 所示。这个过程与本书 14.1 节中描述的过程类似。

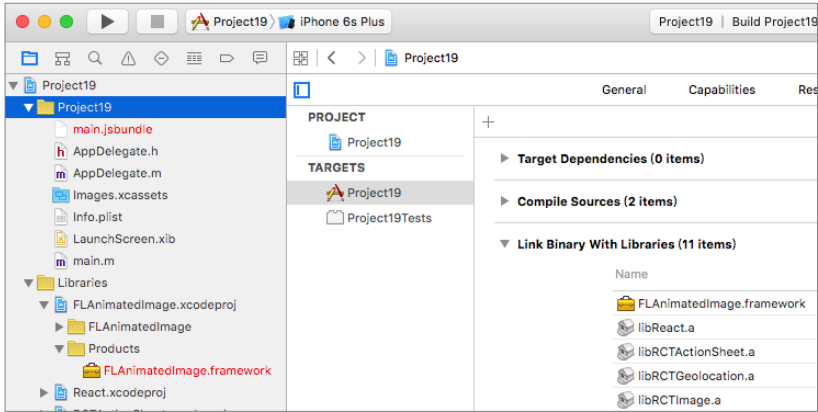


图 17-1 加入 FLAnimatedImage.Xcodeproj 项目文件并链接好

FLAnimatedImage 项目加入后, 还需要将项目文件添加在查找路径中。点击选中项目名称后, 选择“Build Settings”, 向下滚动找到“Search Paths”栏目, 双击选择“Header Search Paths”, 如图 17-2 所示。

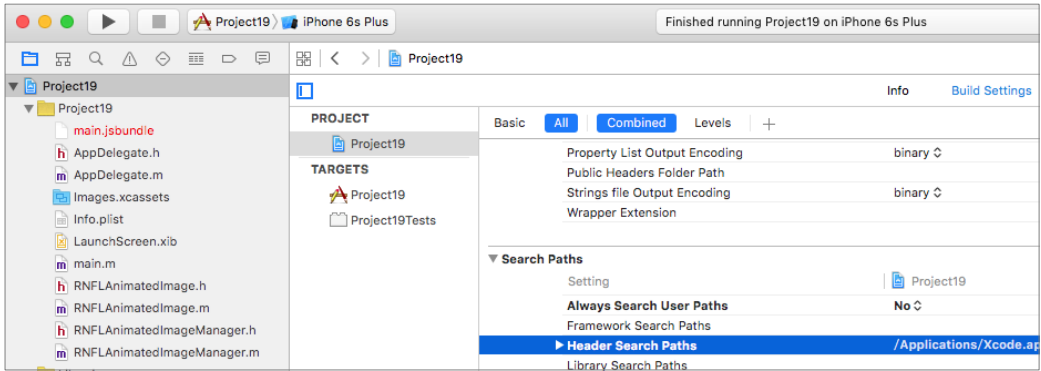


图 17-2 将加入的项目文件添加到查找路径中

双击后, 弹出窗口左下角的“+”号, 在输入框中输入“\$(SRCROOT)/ FLAnimatedImage-master/FLAnimatedImage”, 如图 17-3 所示。

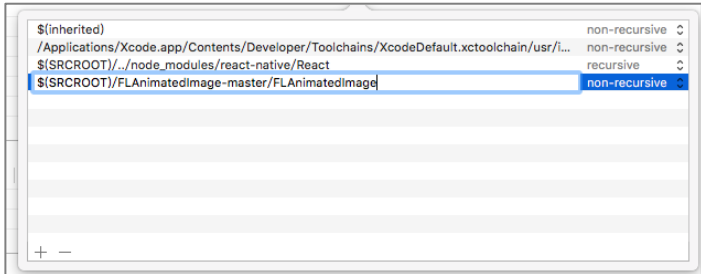


图 17-3 添加查找路径

完成输入后, 在输入框外点击一下鼠标, 结束输入。

开发者还需要将部署目标改为 8.0 或者以上, 如图 17-4 所示。

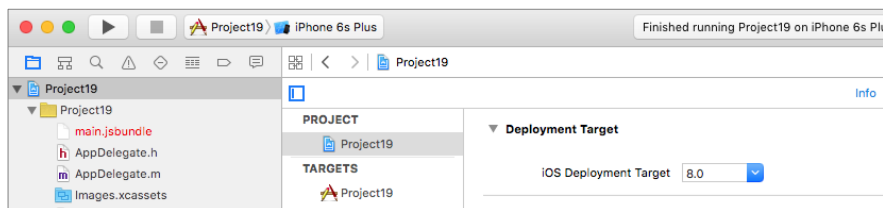


图 17-4 修改部署目标

17.1.2 创建视图管理类

原生代码实现的视图需要由一个 `RCTViewManager` 类的子类创建并管理。这个子类在功能上与视图管理器类似，并且是一个单例（只存在一个实例）。开发者在这个子类中创建原生视图，React Native 框架将这个子类提供给 `RCTUIManager`，`RCTUIManager` 则会反过来委托它在需要时设置、更新视图的属性。它通常还是原生代码视图的委托者，原生代码视图通过它给 JavaScript 发送事件。现在，我们在项目中建立一个 `RCTViewManager` 类的子类——`RNFLAnimatedImageManager` 类。

代码 17-1，RNFLAnimatedImageManager.h:

```
#import "RCTViewManager.h"

@interface RNFLAnimatedImageManager : RCTViewManager
@end
```

代码 17-2，RNFLAnimatedImageManager.m:

```
#import <Foundation/Foundation.h>
#import "RCTBridge.h"
#import "RNFLAnimatedImageManager.h"
#import "RNFLAnimatedImage.h"

@implementation RNFLAnimatedImageManager
RCT_EXPORT_MODULE(); //添加 RCT_EXPORT_MODULE() 标记宏来表明这个类需要被导出

@synthesize bridge = _bridge;

- (UIView *)view
{
    return [[RNFLAnimatedImage alloc] initWithEventDispatcher:self.bridge.eventDispatcher];
}

- (dispatch_queue_t)methodQueue
{
    return dispatch_get_main_queue();
}

RCT_EXPORT_VIEW_PROPERTY(src, NSString); //声明属性
RCT_EXPORT_VIEW_PROPERTY(contentMode, NSNumber);

- (NSArray *) customDirectEventTypes {
    return @[
        @"onFrameChange"
    ];
}

- (NSDictionary *) constantsToExport { //创建跨语言常量
    return @[
        @"ScaleAspectFit": @(UIViewContentModeScaleAspectFit),
        @"ScaleAspectFill": @(UIViewContentModeScaleAspectFill),
    ];
}
```

```

        @"ScaleToFill": @(UIViewContentModeScaleToFill)
    };
}
@end

```

17.1.3 封装开源代码中的视图类

有些原生代码实现的视图类在实现时并没有考虑到会被封装成一个 React Native 组件,因此开发者通常还需要在这些视图类外再封装一层视图类,用来处理与 React Native 相关的消息和事件。

React Native 框架为其对应的原生代码视图类提供了生命周期函数。当有需要时,开发者可以在外层视图类中实现一个或者多个生命周期函数,按业务逻辑对组件进行控制。这些生命周期函数包括:

- `insertReactSubview` 函数,当原生代码视图被要求加入子视图时,此函数会被调用。它的原型是- (void)insertReactSubview:(UIView *)view atIndex:(NSInteger)atIndex。
- `removeReactSubview` 函数,当原生代码视图的某个子视图被移除时,此函数会被调用。它的原型是- (void)removeReactSubview:(UIView *)subview。
- `layoutSubviews` 函数,当原生代码视图将被渲染时,此函数会被调用。它对应 React Native 组件的 `componentWillMount` 事件。它的原型是- (void)layoutSubviews。
- `removeFromSuperview` 函数,当原生代码视图将被卸载时,此函数会被调用。它对应 React Native 侧的 `componentWillUnmount` 事件。它的原型是- (void)removeFromSuperview。

现在我们实现 `RNFLAnimatedImage` 类来封闭真正的原生代码视图。

代码 17-3, `RNFLAnimatedImage.h`:

```

#import "RCTEventDispatcher.h"
#import "RCTView.h"
#import "FLAnimatedImage/FLAnimatedImage.h"
@class RCTEventDispatcher; //导入 React Native 框架的事件调度器
@interface RNFLAnimatedImage : UIView
@property (nonatomic, assign) NSString *src;
@property (nonatomic, assign) NSNumber *contentMode;
- (instancetype)initWithEventDispatcher:(RCTEventDispatcher *)eventDispatcher NS_
DESIGNATED_INITIALIZER;
@end

```

`RNFLAnimatedImage` 类的实现如代码 17-4 所示。

代码 17-4, `RNFLAnimatedImage.m`:

```

#import <Foundation/Foundation.h>
#import "FLAnimatedImageView.h"
#import "RCTBridgeModule.h"
#import "RCTEventDispatcher.h" //导入 React Native 框架的事件调度器头文件
#import "UIView+React.h"
#import "RCTLog.h"
#import "RNFLAnimatedImage.h"
@implementation RNFLAnimatedImage : UIView {
    RCTEventDispatcher *_eventDispatcher;
    FLAnimatedImage *_image; //真正的原生代码相关引用
}

```

```

    FLAnimatedImageView *_imageView;           //真正的原生代码视图引用
}
- (instancetype)initWithEventDispatcher:(RCTEventDispatcher *)eventDispatcher
{
    if ((self = [super init])) {
        _eventDispatcher = eventDispatcher;
        _imageView = [[FLAnimatedImageView alloc] init];    //创建原生代码视图

        [_imageView addObserver:self forKeyPath:@"currentFrameIndex" options:0 context:nil];
    }
    return self;
}
-(void)observeValueForKeyPath:(NSString *)keyPath ofObject:(id)object change:(NSDictionary
<NSString *,id> *)change context:(void *)context {
    if (object == _imageView) {
        if ([keyPath isEqualToString:@"currentFrameIndex"]) {
            [_eventDispatcher sendInputEventWithName:@"onFrameChange" body:@{
                @"currentFrameIndex":[NSNumber numberWithInt:[object currentFrameIndex]],
                @"frameCount": [NSNumber numberWithInt:[_imageView frameCount]],
                @"target": self.reactTag
            }];
        }
    }
}
- (void)removeFromSuperview    //这个函数对应 React Native 侧的 componentWillUnmount 事件
{
    [_imageView removeObserver:self forKeyPath:@"currentFrameIndex"];
    _eventDispatcher = nil;
    [super removeFromSuperview];
}
-(void)reloadImage {
    _image = [FLAnimatedImage animatedImageWithGIFData:[NSData
dataWithContentsOfURL:[NSURL URLWithString:_src]]];
    _imageView.contentMode = [_contentMode integerValue];
    _imageView.animatedImage = _image;
}
- (void)layoutSubviews    //对应 React Native 组件的 componentWillMount 事件
{
    _imageView.frame = self.bounds;
    [self addSubview:_imageView];
}
- (void)setSrc:(NSString *)src
{
    if (![src isEqual:_src]) {
        _src = [src copy];
        [self reloadImage];
    }
}
- (void)setContentMode:(NSNumber *)contentMode
{
    if (![contentMode isEqual:_contentMode]) {
        _contentMode = [contentMode copy];
        [self reloadImage];
    }
}
@end

```

17.1.4 在 React Native 侧调用私有组件

代码 17-5, FLAnimatedImage.js:

```
var React = require('react-native');
var { requireNativeComponent, PropTypes, NativeModules, View } = React;
var {
  ScaleToFill,
  ScaleAspectFit,
  ScaleAspectFill
} = NativeModules.RNFLAnimatedImageManager; //导入在原生代码侧声明的跨语言常量
var MODES = {
  'stretch': ScaleToFill,
  'contain': ScaleAspectFit,
  'cover': ScaleAspectFill
}
var FLAnimatedImage = React.createClass({
  propTypes: { //在 React Native 侧设置私有组件支持的属性
    contentMode: PropTypes.number,
    src: PropTypes.string,
    resizeMode: PropTypes.string,
    onFrameChange: PropTypes.func,
    ...View.propTypes //通过 ES 6 的剩余和延展属性, 声明私有组件
  }, //支持 View 组件的所有属性
  render() {
    var contentMode = MODES[this.props.resizeMode] || MODES.stretch;
    return (
      <RNFLAnimatedImage
        {...this.props}
        contentMode={contentMode}
      />
    );
  },
});
var RNFLAnimatedImage = requireNativeComponent('RNFLAnimatedImage', FLAnimatedImage);
module.exports = FLAnimatedImage;
```

代码 17-6, index.ios.js:

```
'use strict';
var React = require('react-native');
var FLAnimatedImage=require('./FLAnimatedImage');
var {
  AppRegistry,
  StyleSheet,
  Text,
  View,
} = React;

var Project19 = React.createClass({
  render: function() {
    return (
      <View style={styles.container}>
        <FLAnimatedImage style={styles.container}
          src="https://upload.wikimedia.org/wikipedia/commons/2/2c/Rotating_earth_%28large%29.gif"
          resizeMode="contain" >
```

```

        </FLAnimatedImage>
        <Text style={styles.textStyle}>
            Hello, world.
        </Text>
    </View>
    );
}
});
var styles = StyleSheet.create({
    container: {
        flex:1,
    },
    container1: {
        width:300,
        height:300,
        justifyContent: 'center',
        alignItems: 'center',
        backgroundColor: '#F5FCFF',
    },
    textStyle:{
        position:'absolute',
        top:320,
        left:50,
        height:40,
        fontSize:30,
        backgroundColor:'transparent'
    }
});
AppRegistry.registerComponent('Project19', () => Project19);

```

17.1.5 例程运行效果

图 17-5 展示了例程运行效果。图中的 Hello World 是为了演示如何在私有组件上叠加其他组件。在手机上运行时，会显示一个不停转动的地球。



图 17-5 例程运行效果

17.2 使用 Swift 语言创建私有的 React Native 组件

本节将示范如何将开源的使用 Swift 语言开发的 GradientView 项目封装成一个 React Native 组件，供开发者在 React Native 开发中调用。GradientView 可以用来生成一个颜色渐变的视图，其显示效果如图 17-6 所示（因为本书是黑白印刷，所以显示的是从黑到白的渐变图，在手机上可以实现任意三种颜色的渐变）。

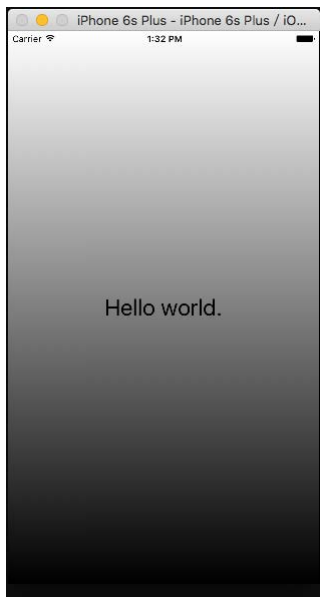


图 17-6 颜色渐变的视图效果

这种渐变的背景色不是使用一张背景图片实现的，而是使用代码来实现的，这对于减少项目复杂度和安装包大小都有帮助。

17.2.1 整合开源项目

首先开发者需要下载开源项目。项目的下载地址是 <https://github.com/soffes/GradientView>。下载后解压项目文件，并将解压得到的 GradientView-master 文件夹直接移动到 Project18 项目目录下的 iOS 子目录中。

然后进入 Xcode，在项目文件列表上单击鼠标右键，选择“Add files to ‘Project18’ ...”，在文件选择框中选择 GradientView-master 目录下的 Gradient View.Xcodeproj 文件，单击“Add”按钮确定。成功整合后的项目目录如图 17-7 所示。

在整合后的项目结构中，Gradient View 下的 Example 目录是原来开源项目的例程目录，而 Products 目录是例程编译后生成的可执行文件目录。它们对当前示例不会产生影响，因此不用删除它们。

接下来需要将编译得到的 GradientView.framework 与项目库链接起来。

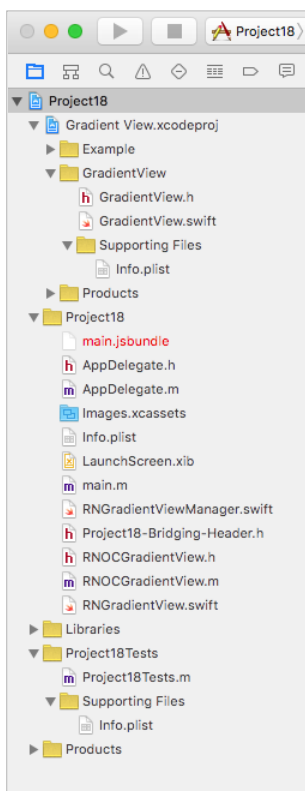


图 17-7 整合 GradientView 项目后的项目目录

选择项目列表最上方的 Project18, 然后点击“Build Phases”, 展开“Link Binary With Libraries”列表, 点击 “+”, 如图 17-8 所示。

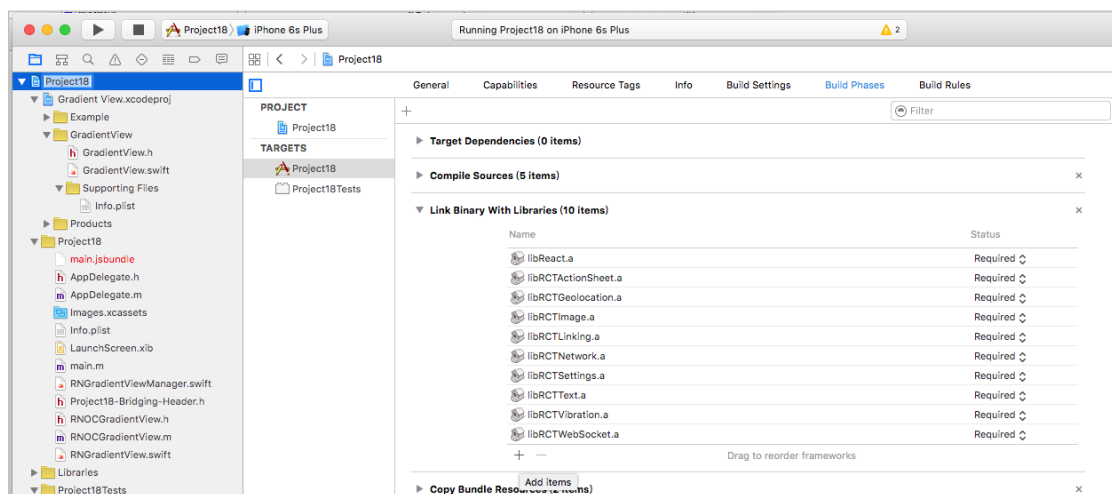


图 17-8 将编译得到的框架与项目库链接

打开添加框架和库对话框, 如图 17-9 所示。

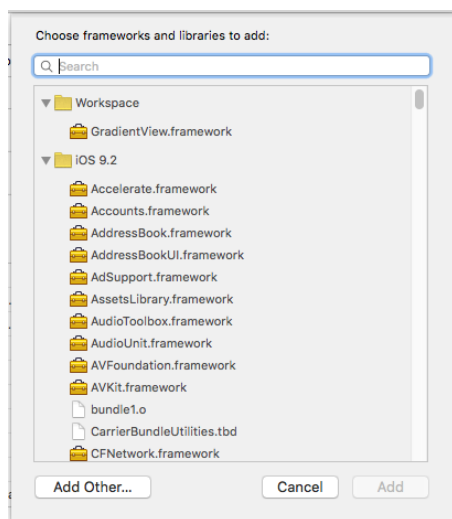


图 17-9 添加框架和库对话框

选择 GradientView.framework，然后单击“Add”按钮，完成添加，如图 17-10 所示。

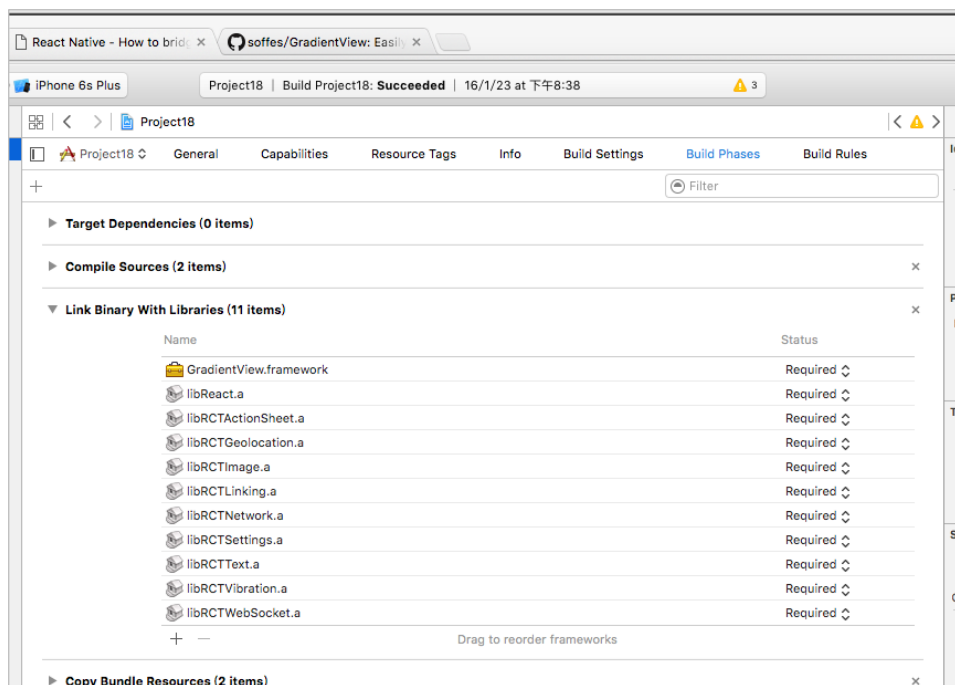


图 17-10 完成添加框架和库

添加完成后，还需要指明添加框架的搜索路径。点击“Build Settings”，再单击“All”按钮，然后向下滚动主窗口直到看到“Search Paths”栏目，在该栏目下的“Framework Search Paths”中输入：\$(SRCROOT)/ GradientView-master，如图 17-11 所示。

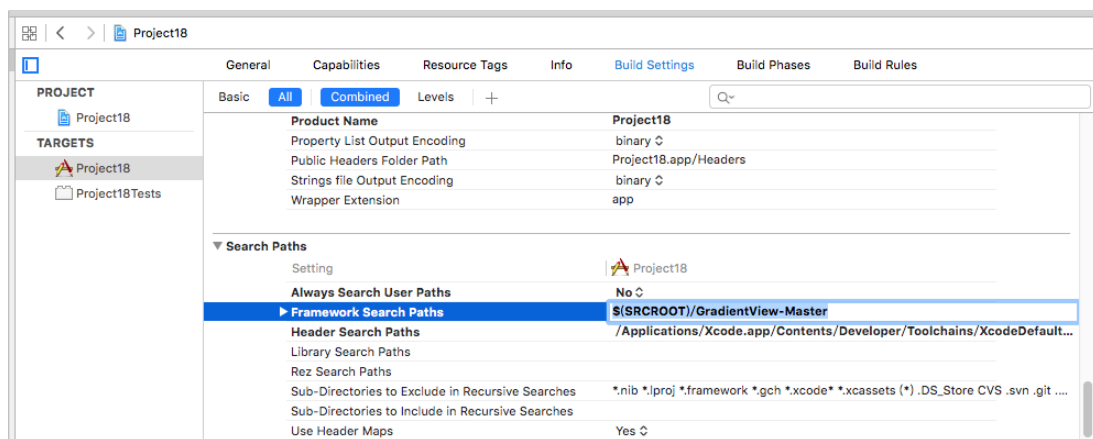


图 17-11 添加框架的搜索路径

输入完成后回车，“Framework Search Paths”会变成对应的绝对路径。

17.2.2 建立组件管理者和桥接头文件

首先开发者需要建立一个新的 Swift 文件。

选择项目文件列表中的 Project18 并单击鼠标右键，选择“New File...”，打开如图 17-12 所示的对话框。

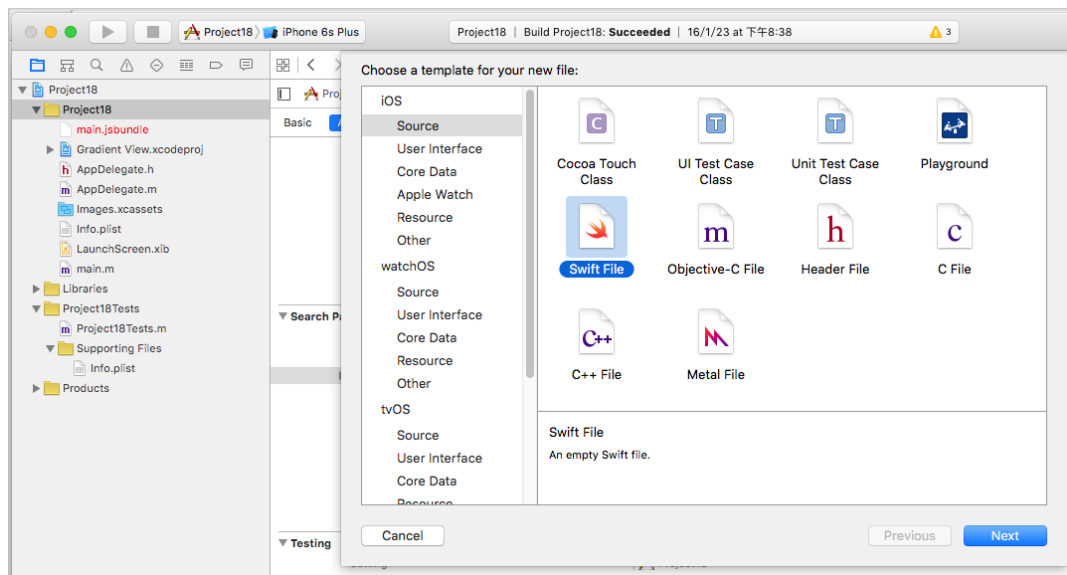


图 17-12 新建 Swift 文件步骤一

选择“Swift File”，单击“Next”按钮，打开如图 17-13 所示的对话框。

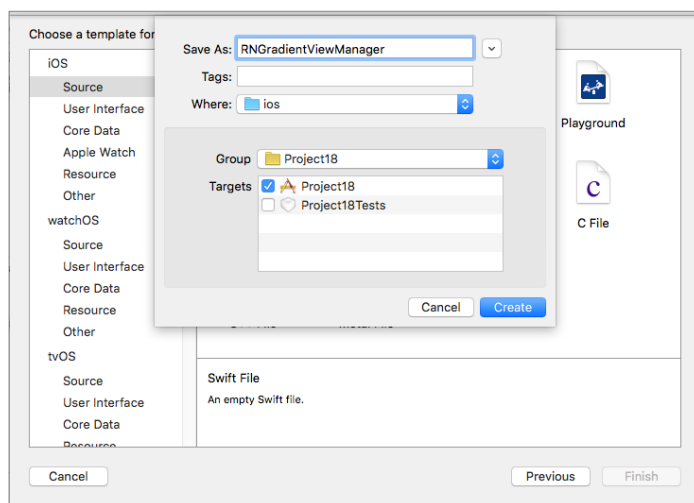


图 17-13 新建 Swift 文件步骤二

输入文件名：RNGradientViewManager，单击“Create”按钮，打开如图 17-14 所示的对话框。

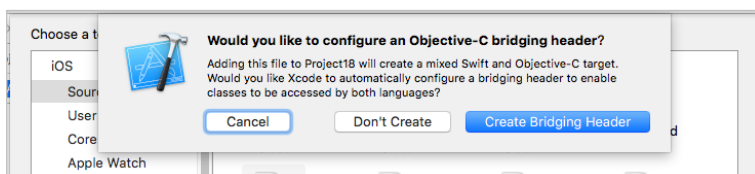


图 17-14 新建 Swift 文件步骤三

这个对话框询问你是否配置一个 Objective-C 的桥接头文件。单击“Create Bridging Header”按钮，让 Xcode 为我们自动建立一个桥接头文件。这个文件将会被 Xcode 自动命名，在本例中，它的名称是 Project18-Bridging-Header.h，如图 17-15 所示。注意，这个文件的名称开发者不能自己更改。

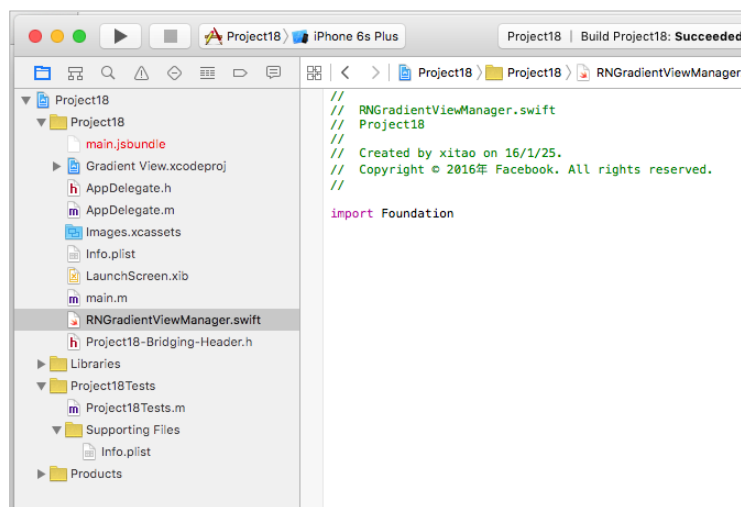


图 17-15 新建 Swift 文件步骤四

我们先不修改所建立的 Swift 文件，而是修改 Project18-Bridging-Header.h 文件。在 Xcode 自动建立的这个文件中加入一行，如代码 17-7 所示。

代码 17-7:

```
//
// Use this file to import your target's public headers that you would
// like to expose to Swift.
#import "RCTViewManager.h" //需要读者手动添加的代码
```

17.2.3 Objective-C 与 React Native 接口部分

虽然原生组件是用 Swift 语言编写的，但想要将原生组件封装成 React Native 可以使用的私有组件，还是需要使用 Objective-C 语言来处理与 React Native 框架的接口部分。

建立一个 Objective-C 语言的 RNOCGradientView 类（创建步骤参考本书 4.1 节，这里不再重述）。

Xcode 将为我们建立两个文件：RNOCGradientView.h 和 RNOCGradientView.m。项目列表如图 17-16 所示。

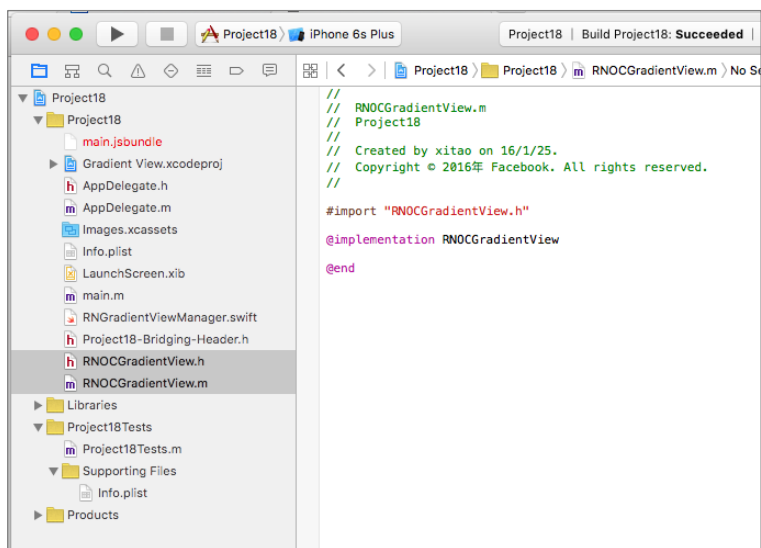


图 17-16 创建后的项目列表

编辑 RNOCGradientView.m 文件，在 React Native 开发中，声明我们导出的组件名称为 RNGradientViewSwift，并且声明该组件需要两个属性：locations 和 colors。它们的类型都是数组，如代码 17-8 所示。

代码 17-8, RNOCGradientView.m:

```
#import "RNOCGradientView.h"
#import "RCTViewManager.h"
// RCT_EXTERN_MODULE 接口定义了组件名称为 RNGradientViewSwift
@interface RCT_EXTERN_MODULE(RNGradientViewSwift, RCTViewManager)
//声明实现 View 必需的两个属性
```

```
RCT_EXPORT_VIEW_PROPERTY(locations, NSArray);
RCT_EXPORT_VIEW_PROPERTY(colors, NSArray);

@end
```

RNOCGradientView.h 文件内容如代码 17-9 所示。

代码 17-9, RNOCGradientView.h:

```
#import "RCTView.h"
@interface RNOCGradientView : RCTView

@property (nonatomic, assign) NSArray *locations;
@property (nonatomic, assign) NSArray *colors;

@end
```

我们导入 React Native 框架定义的 RCTView 头文件，然后定义两个类成员属性。

17.2.4 使用 Swift 语言实现组件控制

在本例程中，RNGradientViewManager 类的 view 函数需要返回一个 View，这个 View 由开发者定义并实现。现在建立一个名为 RNGradientView 的 Swift 文件以定义 RNGradientView 类——RNGradientViewManager 类的 view 函数返回的对象类型。修改后的 RNGradientView.swift 文件内容如代码 17-10 所示。

代码 17-10, RNGradientView.swift:

```
import Foundation
import GradientView

@objc(RNGradientView)
class RNGradientView : GradientView {

    override init(frame: CGRect) {
        super.init(frame: frame);
        self.frame = frame;
    }

    required init?(coder aDecoder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }

    func setLocations(locations: NSArray) {
        self.locations = locations.map({ return $0 as! CGFloat});
    }

    func setColors(colors: NSArray) {
        self.colors = colors.map({return RCTConvert.UIColor($0)})
    }
}
```

建立文件后，会显示编译无法通过。需要对项目设置进行修改，但我们先不修改，输入最后一份 Swift 代码后再进行修改。

修改开始时建立的空的 RNGradientViewManager.swift 文件，将其修改为代码 17-11。

代码 17-11, RNGradientViewManager.swift:

```
import Foundation
@objc(RNGradientViewSwift)
class RNGradientViewManager : RCTViewManager {
    override func view() -> UIView! {
        return RNGradientView();
    }
}
```

现在修改项目编译设置。首先点击项目列表最上方的 Project18 项目图标,然后选择 TARGETS 列表中的 Project18,再选择 General,最后修改“Deployment Info”中的“Deployment Target”,让它等于或者高于 8.0,如图 17-17 所示。

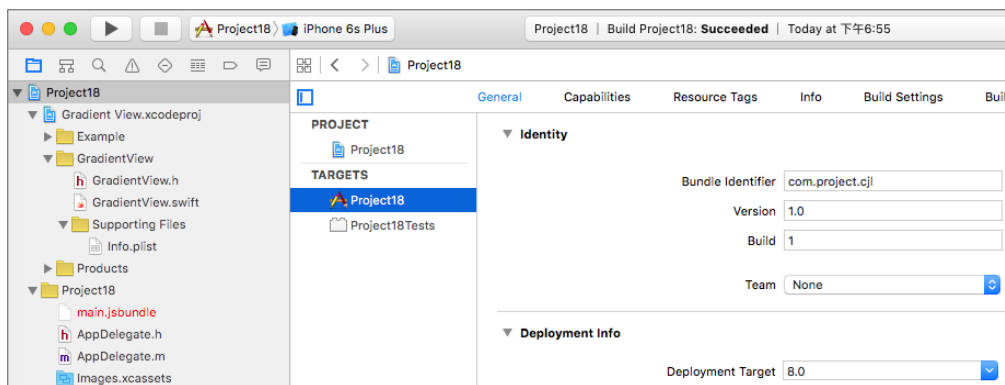


图 17-17 修改部署目标

在“Build Phases”中，在“Target Dependencies”下通过点击“+”添加 GradientView，如图 17-18 所示。

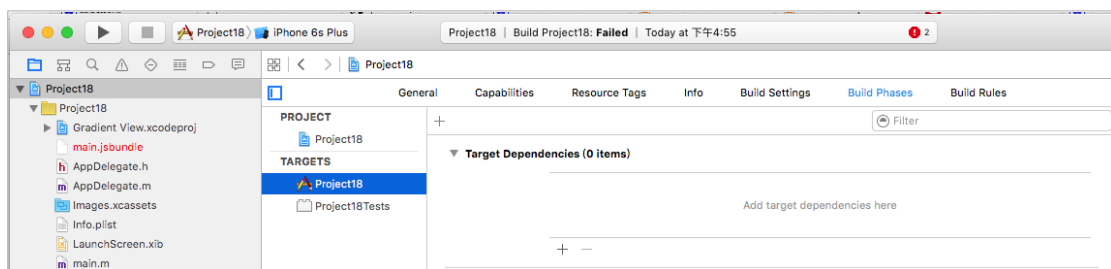


图 17-18 添加 GradientView 步骤一

点击“+”后，在弹出框中选择 GradientView，单击“Add”按钮，如图 17-19 所示。

至此，新建立的 RNGradientView.swift 可以编译通过。

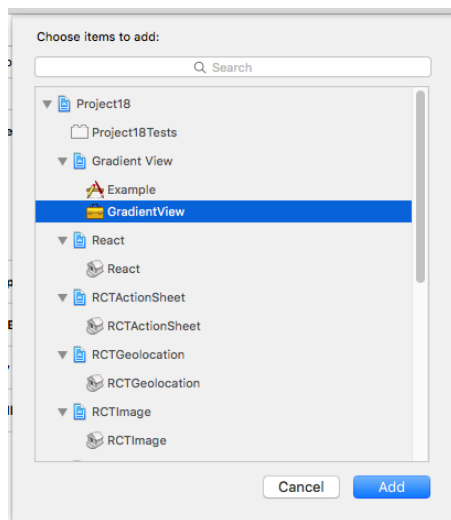


图 17-19 添加 GradientView 步骤二

17.2.5 在 React Native 侧调用私有组件

现在我们已经通过 Swift 语言和 Objective-C 语言在原生代码侧实现了 RNGradientViewSwift 私有组件供 React Native 侧调用。下面介绍在 React Native 侧调用实现的私有组件。

在项目文件夹下新建一个 JavaScript 文件，命名为 GradientView.js。它的内容见代码 17-12。

代码 17-12, GradientView.js:

```
//导入 requireNativeComponent、processColor 函数
import React, { requireNativeComponent, processColor } from 'react-native';
//导入私有组件
let RNGradientView = requireNativeComponent('RNGradientViewSwift', GradientView);
//定义 JavaScript 侧的私有组件
class GradientView extends React.Component {
  render() {
    let { colors, ...otherProps } = this.props;
    return <RNGradientView {...otherProps} colors={processColor(colors)} />;
  }
}
GradientView.propTypes = { //属性声明
  colors: React.PropTypes.array.isRequired,
  locations: React.PropTypes.array,
}
module.exports = GradientView; //导出组件
```

JavaScript 侧的私有组件的使用示例见代码 17-13。

代码 17-13, index.ios.js:

```
'use strict';
let GradientView = require('./GradientView'); //导入代码 17-12 中定义的私有组件
var React = require('react-native');
var {
  AppRegistry, StyleSheet, Text, View
```

```

} = React;
var Project18 = React.createClass({
  render: function() {
    return (
      <GradientView style={styles.gradient}
        locations={[0, .5, 1.0]}
        colors={['white', 'grey', 'black']}>
        <Text style={styles.textStyle}>
          Hello world.
        </Text>
      </GradientView>
    );
  }
});
var styles = StyleSheet.create({
  gradient: { //私有组件样式定义
    flex:1,
    justifyContent: 'center',
    alignItems:'center'
  },
  textStyle: {
    fontSize:30,
    backgroundColor:'transparent'
  }
});
AppRegistry.registerComponent('Project18', () => Project18);

```

开发者可以对 GradientView 私有组件指定任何 View 的样式键值，GradientView 都可以正确显示这些样式。

17.2.6 例程运行效果

代码 17-13 的运行效果如图 17-6 所示。因为本书黑白印刷，所以指定颜色渐变顺序为 colors={['white', 'grey', 'black']}，读者在电脑上验证时，建议将这句改为：colors={['red', 'yellow', 'blue']}。

17.3 使用 Android SDK 创建私有的 React Native 组件

React Native 可以轻松地将 Android 原生代码实现的 UI 组件封装成 React Native 组件，然后直接在 React Native 代码中使用。

在本节中，我们将把 KenBurnsView 这个开源的 Android UI 组件封装成一个 React Native 组件，并在 React Native 代码中调用它。

KenBurnsView 组件能够显示一张图片的一部分，然后通过改变显示位置来显示其他部分。配上合适的图片，就能产生一种乘坐在直升机上俯视某区域的视觉效果。它的开源地址是 <https://github.com/flavioarfaria/KenBurnsView>，对它感兴趣的读者可以自行去下载其文档与代码。但在本例程中，我们不需要下载任何东西。

17.3.1 准备原生代码 UI 组件

建议读者从备份的项目文件压缩包中恢复出一份初始的项目文件夹以继续下面的讨论。

在开发前，我们需要一张图片。在百度图片中搜索“北京俯瞰图”，然后随便下载一张北京俯瞰图。将这个图片命名为 Beijing.jpg，放在“项目文件夹”/android/app/src/main/assets 目录下（如果没有 assets 目录请自行建立）。

在 Android Studio 中打开 Android 工程（方法在 4.2 节中讨论过，这里不再重述）。

如果是开发者自己实现的原生代码 UI 组件，那么这时，应当将 UI 组件的原生代码集成在项目中。这个组件必须是 Android SDK 中 View 类的子类，或者是 ViewGroup 的子类。如果是 ViewGroup 的子类，React Native 框架将允许在封装的组件中包含其他组件。

在本例中，我们使用在开源项目中已经实现好的类。打开应用的 build.gradle 文件，如图 17-20 所示。

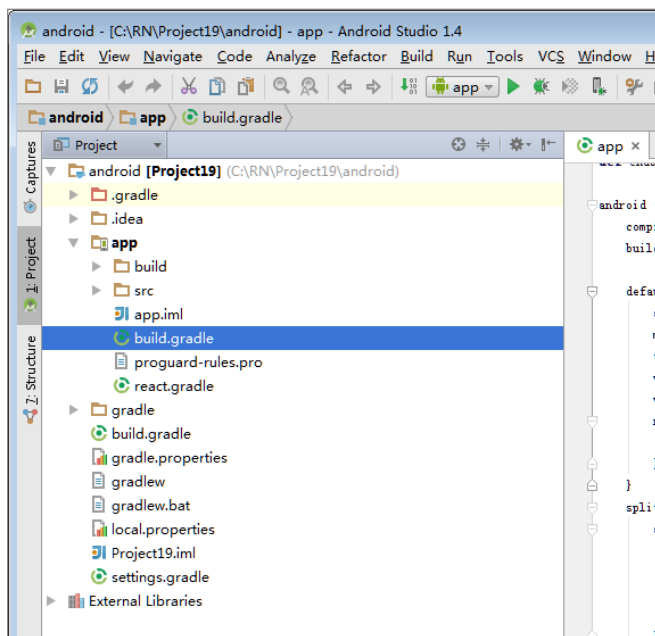


图 17-20 打开 build.gradle 文件

build.gradle 最后的部分如代码 17-14 所示，在文件尾部添加要求编译的开源软件包。

代码 17-14，build.gradle 结束部分：

```
.....
dependencies {
    compile fileTree(dir: "libs", include: ["*.jar"])
    compile "com.android.support:appcompat-v7:23.0.1"
    compile "com.facebook.react:react-native:0.19.+"
    compile "com.flaviofaria:kenburnsview:1.0.7" //此行是开发者需要添加的内容
}
```


17.3.2 实现原生 UI 管理类

在项目文件夹中新建 `KenBurnsViewManager.Java` 文件，文件内容见代码 17-15。

代码 17-15, `KenBurnsViewManager.Java`:

```
package com.Project19;
import com.flaviofaria.kenburnsview.KenBurnsView;
import com.facebook.react.uimanager.SimpleViewManager;
import com.facebook.react.uimanager.ThemedReactContext;
import com.facebook.react.uimanager.ReactProp;
import java.io.InputStream;
import android.graphics.drawable.Drawable;
import android.util.Log;
//在下面一行代码中，我们通过 extends SimpleViewManager<KenBurnsView>指明
//原生 UI 管理类用来管理的原生代码 View 子类的类型
public class KenBurnsViewManager extends SimpleViewManager<KenBurnsView> {
    public static final String REACT_CLASS = "KenBurnsView";
    private ThemedReactContext aContext;
    @Override // getName 方法返回的名字会被用在 JavaScript 端引用这个原生视图
    public String getName() {
        return REACT_CLASS;
    }

    @Override
    protected KenBurnsView createViewInstance(ThemedReactContext context) {
        aContext = context; //保存创建时传入的上下文实例以备后用
        KenBurnsView aView = new KenBurnsView(aContext); //创建原生 View
        try { //对 View 进行初始化，读入图片数据
            InputStream ims = aContext.getAssets().open("beijing.jpg");
            Drawable aDrawable = Drawable.createFromStream(ims, null);
            aView.setImageDrawable(aDrawable);
        } catch (Exception error) {
            Log.e("RNMessage", "KenBurnsView error:" + error);
        }
        return aView; //返回创建好的原生 View 实例给 React Native 框架
    }

    //下面的函数用来对传入的属性进行处理，每个需要处理的属性
    //都要有一个这样的函数。本例中，我们只自行定义并处理了一个
    //属性，其他属性交给 React Native 按默认处理
    @ReactProp(name = "imgSource")
    public void setSource(KenBurnsView view, String source) {
        try {
            InputStream ims = aContext.getAssets().open(source);
            Drawable aDrawable = Drawable.createFromStream(ims, null);
            view.setImageDrawable(aDrawable);
        } catch (Exception error) {
            Log.e("KenBurnsView", "setSource", error);
        }
    }
}
```

代码输入完成后, `KenBurnsView` 类会被显示成红色, 表示无法找到它的声明。这时选择 Android Studio 菜单中的 “Build” → “Rebuild Project”, 重新编译项目后, 红色错误提示就消失了。

17.3.3 创建原生 UI 实例

在代码 17-15 中实现的 `createViewInstance` 函数用来进行原生 UI 实例的创建和初始化。原生 UI 实例被创建后，不需要保存一份引用在本管理类中，就如同代码 17-5 中实现的那样。当需要对这个原生 UI 进行处理时，React Native 框架会再次向开发者提供它的引用。

17.3.4 实现对属性的支持

在原生 UI 管理类中，开发者可以设置本属性需要使用哪些属性，以及实现当 React Native 侧设置了这些属性后，原生代码的处理流程。每个属性都需要实现一个带有 `@ReactProp`（或 `@ReactPropGroup`）注解的设置函数。函数的第一个参数是原生 UI 的引用，第二个参数是要设置的属性值。函数的访问控制必须被声明为 `public`，并且不需要有返回值。React Native 侧属性类型会由相应函数的第二个参数类型自动决定，支持的类型有：`boolean`、`int`、`float`、`double`、`String`、`Boolean`、`Integer`、`ReadableArray`、`ReadableMap`。

`@ReactProp` 注解必须包含一个字符串类型的参数 `name`，这个参数指定了对应属性在 JavaScript 端的名字。

除了 `name`，`@ReactProp` 注解还可以接收 `defaultBoolean`、`defaultInt`、`defaultFloat` 参数。它们必须是对应的基础类型的值（也就是 `boolean`、`int`、`float`），它们会被传递给 `setter` 方法，用于在 React Native 侧没有设置相应属性时提供默认值。注意“默认值”只对基本类型生效，对于其他类型的属性而言，`null` 会作为默认值被提供。

在代码 17-15 中，只实现了对一个属性的处理，允许在 React Native 侧重新指定图片名称。

React Native 框架实现了自定义组件的大部分默认属性支持。我们将在 17.3.7 节中看到。

17.3.5 建立原生 UI 包

开发者需要建立原生 UI 管理类。在项目文件中建立 `KenBurnsViewPackage` 类，它的代码如代码 17-16 所示。

代码 17-16, `KenBurnsViewPackage.java`:

```
package com.project19;
import com.facebook.react.ReactPackage;
import com.facebook.react.bridge.JavaScriptModule;
import com.facebook.react.bridge.NativeModule;
import com.facebook.react.bridge.ReactApplicationContext;
import com.facebook.react.uimanager.ViewManager;
import java.util.Collections;
import java.util.List;
import java.util.Arrays;
public class KenBurnsViewPackage implements ReactPackage {
    @Override
    public List<NativeModule> createNativeModules(ReactApplicationContext reactContext) {
        return Collections.emptyList();
    }
}
```

```

@Override
public List<ViewManager> createViewManagers( ReactApplicationContext reactContext) {
    return Arrays.<ViewManager>asList( new KenBurnsViewManager());
    //原生 UI 管理类在这里被创建并被保存
}
@Override
public List<Class<? extends JavaScriptModule>> createJSMODULES() {
    return Collections.emptyList();
}
}

```

17.3.6 注册原生 UI 管理类

Android 原生开发侧需要实现的最后一步是在 MainActivity 中注册原生 UI 管理类。修改 MainActivity.java，如代码 17-17 所示。

代码 17-17， MainActivity.java:

```

package com.project19;
import com.facebook.react.ReactActivity;
import com.facebook.react.ReactPackage;
import com.facebook.react.shell.MainReactPackage;
import java.util.Arrays;
import java.util.List;
public class MainActivity extends ReactActivity {
    /* Returns the name of the main component registered from JavaScript.
     * This is used to schedule rendering of the component. */
    @Override
    protected String getMainComponentName() {
        return "Project19";
    }
    /* Returns whether dev mode should be enabled.
     * This enables e.g. the dev menu. */
    @Override
    protected boolean getUseDeveloperSupport() {
        return BuildConfig.DEBUG;
    }
    /* A list of packages used by the app. If the app uses additional views
     * or modules besides the default ones, add more packages here. */
    @Override
    protected List<ReactPackage> getPackages() {
        return Arrays.<ReactPackage>asList(
            new MainReactPackage(),                //需要开发者添加的一个逗号
            new KenBurnsViewPackage()              //需要开发者添加的语句
        );
    }
}

```

17.3.7 对应的 React Native 侧实现

接下来，我们在 React Native 侧建立对应的私有组件 JavaScript 文件。KenBurnsView.js 文件内容如代码 17-18 所示。

代码 17-18, KenBurnsView.js:

```
var { requireNativeComponent, PropTypes, View } = require('react-native');
var iface = {
  name: 'KenBurnsView',
  propTypes: {
    imgSource: PropTypes.string,
    ...View.propTypes //通过 ES 6 的剩余和延展属性, 声明 KenBurnsView 组件
  }, //支持 View 组件的所有属性
};
module.exports = requireNativeComponent('KenBurnsView', iface);
```

至此, KenBurnsView 组件已经可以供开发者使用了。代码 17-19 示范了对它的简单使用。

代码 17-19, index.android.js:

```
'use strict';
var React = require('react-native');
var KenBurnsView = require('./KenBurnsView');
var {
  AppRegistry, StyleSheet, Text, View,
} = React;
var Project19 = React.createClass({
  render: function() {
    return (
      <View style={styles.container}>
        <KenBurnsView source='image.jpg' //调用私有组件
          style={styles.container1}/>
        <Text style={styles.welcome}>
          Hello, world
        </Text>
      </View>
    );
  }
});
var styles = StyleSheet.create({
  container: {
    flex:1,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#F5FCFF',
  },
  container1: { //私有组件样式设定
    width:360,
    height:616
  },
  welcome: {
    position:'absolute',
    top:150,
    left:20,
    fontSize: 40,
    textAlign: 'center',
  }
});
AppRegistry.registerComponent('Project19', () => Project19);
```

17.3.8 运行俯视视图例程

代码 17-19 的运行效果如图 17-21 所示。图中的 Hello World 是为了演示如何在这种私有组件上叠加其他组件。在手机上运行时，图片会随机缓慢移动、随机缓慢缩放，产生一种视觉移动的效果。

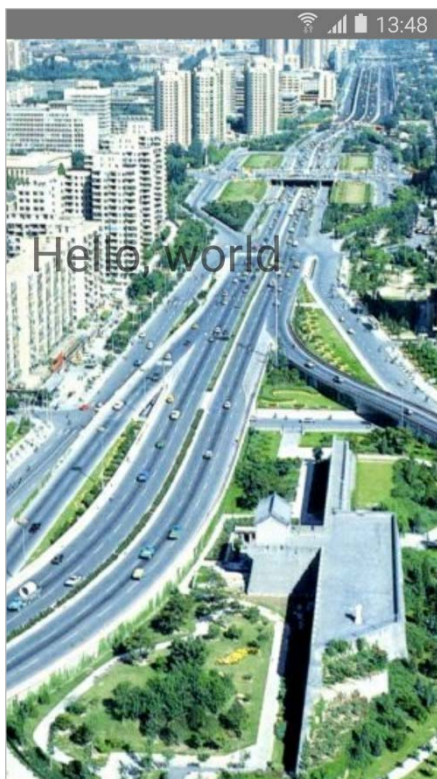


图 17-21 代码 17-19 运行效果

第 18 章

项目配置、生成发布版本安装包及其他

当一个 React Native 应用开发完成后，开发者需要做的最后一个工作就是对其进行配置，然后编译，通过各平台的发布渠道发布安装包，最终让用户能够在自己的手机上下载、安装并使用。

18.1 iOS 平台项目配置

为了修改 React Native 初始化项目的应用名称，打开项目的“Build Settings”，找到“Product Name”项，修改对应的值。修改界面如图 18-1 所示。

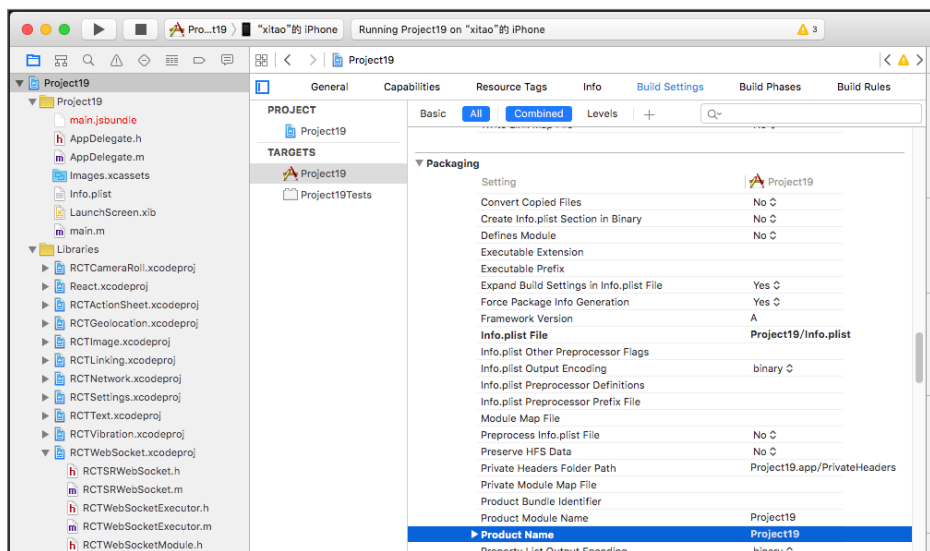


图 18-1 修改应用名称

为了修改应用的显示图标，在项目文件目录中找到 Images.scassets（在 Xcode 新项目中可能叫 Assets.xcassets），选中后再选择该目录下的 AppIcon，这时中间窗口会列出需要哪些图标，点击每一个图标，右侧窗口都会有这个图标要求的详细说明。界面如图 18-2 所示。按照图标要求，将与之像素匹配的图片拖拽到相应的方框即可设置该应用图标。

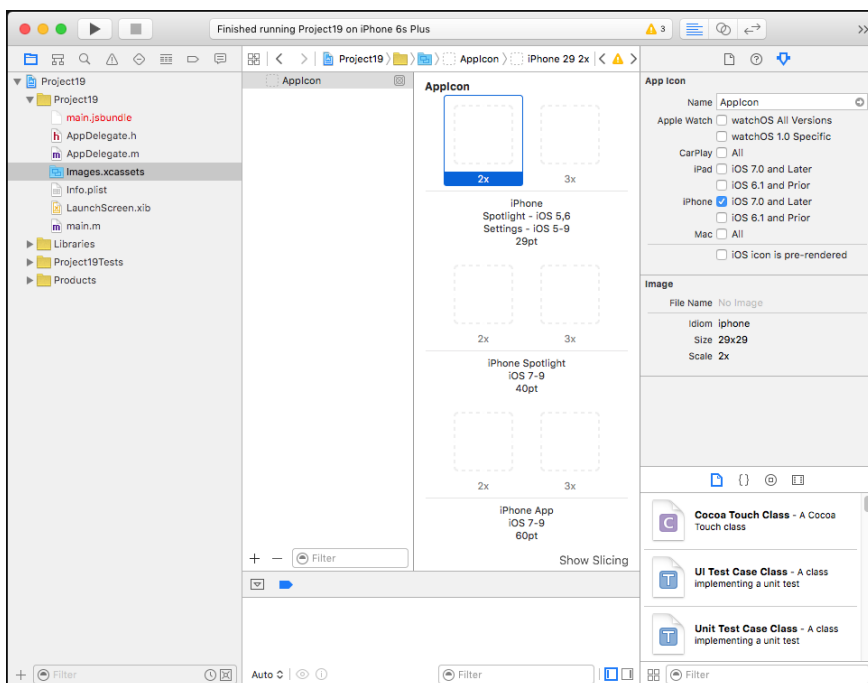


图 18-2 修改应用图标

为了修改应用的启动画面，在项目文件目录中找到 `Images.xcassets`（在 Xcode 新项目中可能叫 `Assets.xcassets`），选中后在该目录下的 `AppIcon` 的空白处单击右键，选择“App Icons & Launch Images”下的“New iOS Launch Image”。界面如图 18-3 所示。

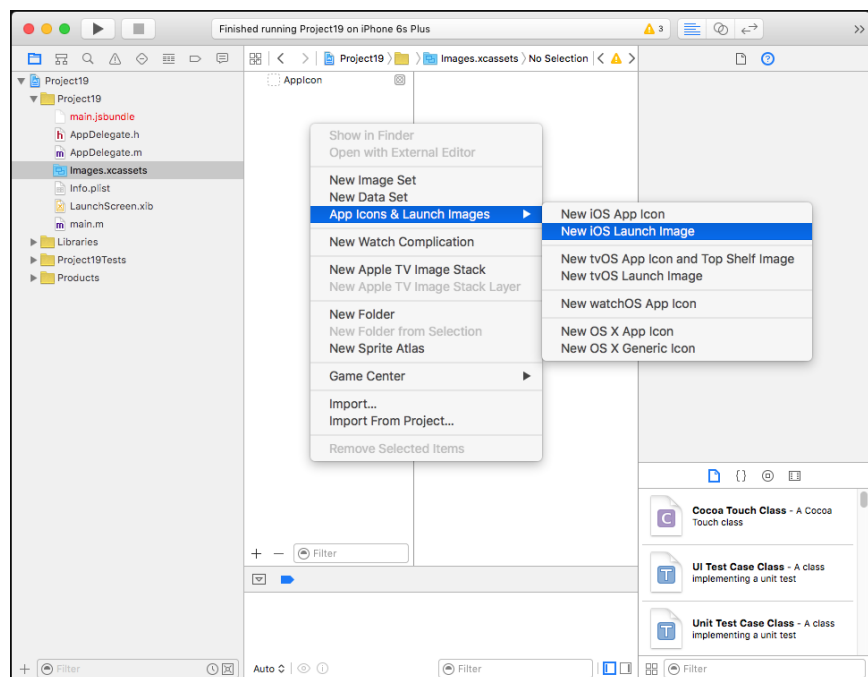


图 18-3 新建启动画面界面

新建启动画面成功后的界面如图 18-4 所示。与设置应用图标相似，点击每一张启动图片，右侧窗口中都会显示相应的图片文件要求说明。右侧窗口中的选项还可以让开发者根据项目需要设置应用不同版本、不同设备的横向和纵向启动图片。其中，Landscape 是横向图片，Portrait 是纵向图片。把准备好的图片文件拖动到相应位置即可。

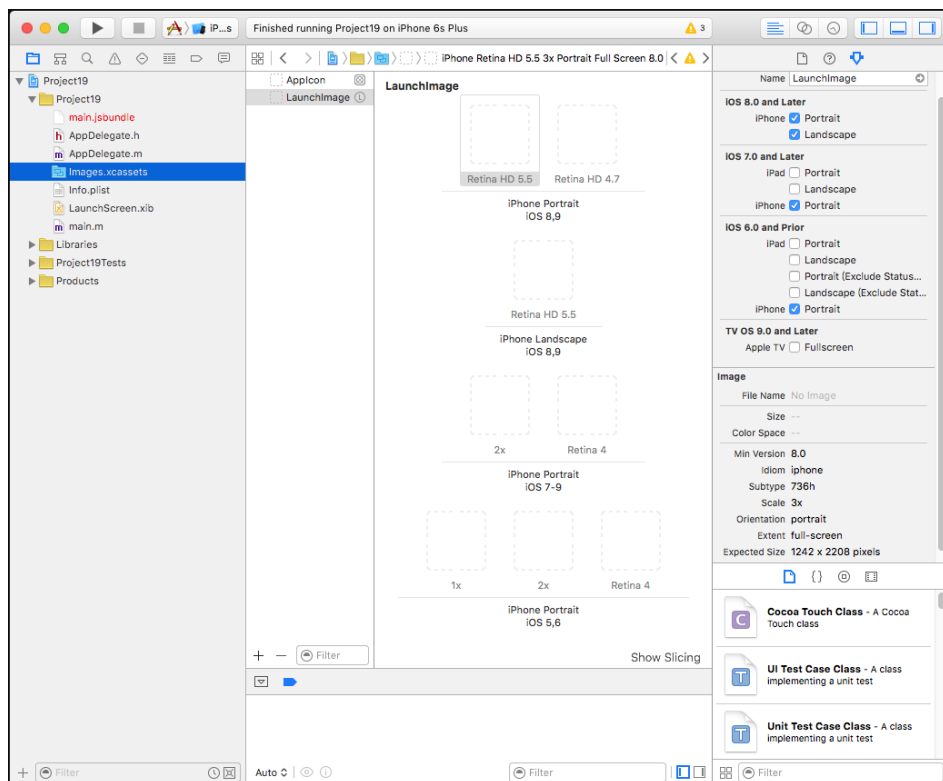


图 18-4 提交启动画面图片文件

启动图片提交好后，选择项目目录中的项目名，在 TARGETS 中选择项目（不要选择下面的项目测试），然后选择上方的“General”，在右侧窗口找到“App Icons and Launch Images”栏目下的“Launch Screen File”，删除方框中的 LaunchScreen。操作如图 18-5 所示。

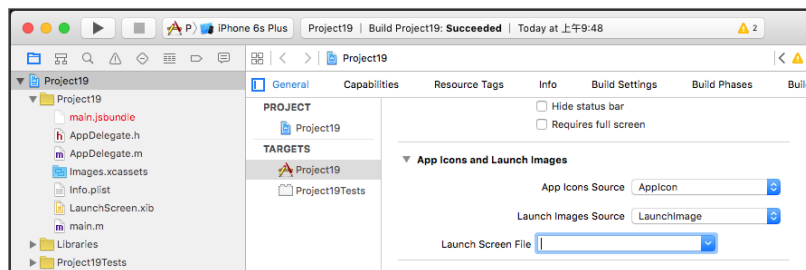


图 18-5 删除默认的启动画面

接下来选择“Launch Images Source”中的“Use Asset Catalog”，在弹出的“Migrate launch images to an asset catalog”对话框中单击“Migrate”按钮，如图 18-6 所示。

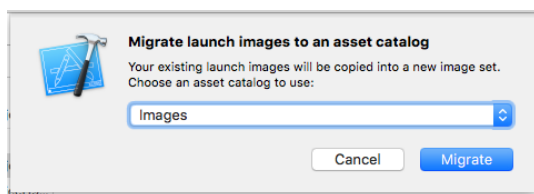


图 18-6 合并启动图片

再单击“Launch Images Source”下拉按钮，选择“LaunchImage”，如图 18-7 所示。

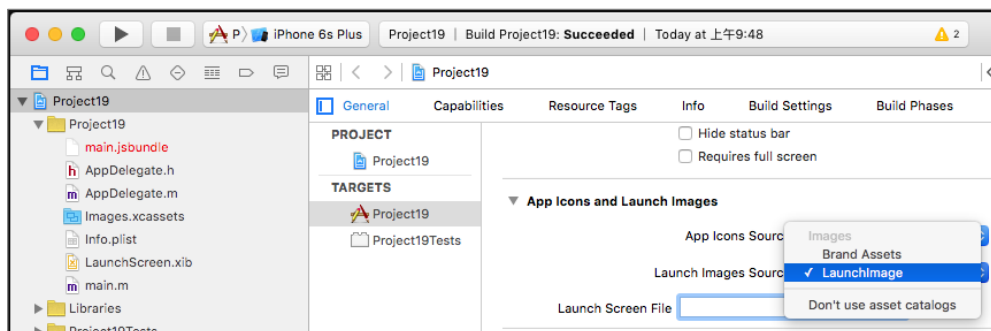


图 18-7 选择启动图片

现在开发者需要返回 Images.xcassets (或者是 Assets.xcassets)，删除多余的“Brand Assets”。删除操作如图 18-8 所示。

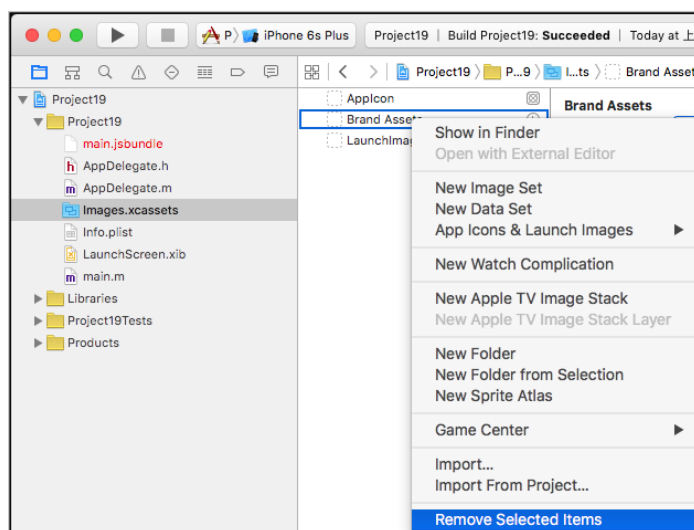


图 18-8 删除“Brand Assets”

如果开发者需要设置应用显示方式（默认是跟随设备放置方式而改变），在项目文件列表中点击项目名，找到并选择 TARGETS 中的项目，然后选择上方的“General”，在右侧窗口中找到“Deployment Info”栏目下的“Device Orientation”（默认勾选了三个方向），开发者可以根据项目需要勾选方向。操作如图 18-9 所示。

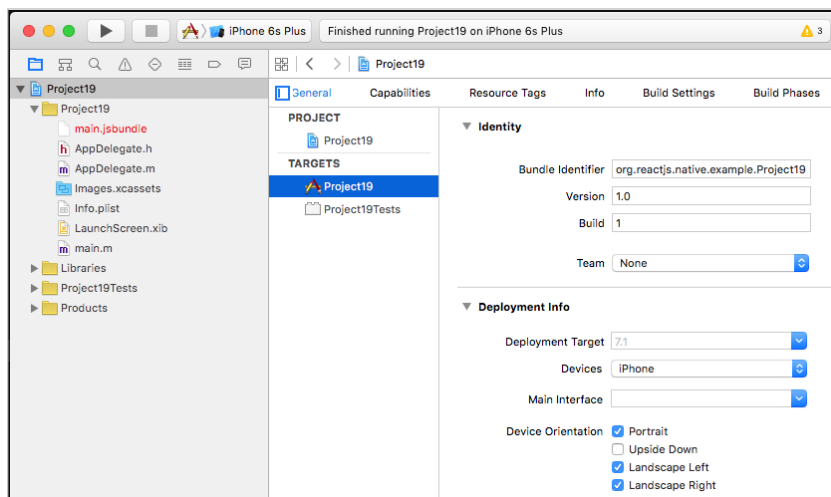


图 18-9 修改应用显示方式

18.2 iOS 平台应用发布

iOS 平台要求开发者完成开发后，将开发的软件包使用苹果官方规定的各种证书（通常都是使用苹果开发者账号按苹果规定生成的）签名，然后提交给苹果审查。苹果审查通过后，应用将出现在苹果应用商店供苹果手机用户下载使用。

使用 React Native 框架开发的 iOS 平台移动应用也要遵循苹果的规定。这个流程超出了本书的讨论范围，这里不再阐述。

18.3 Android 平台项目配置

在正式生成发布版本安装包之前，开发者通常都需要修改应用名称和应用图标，有时还需要修改项目编译的 SDK 版本号和编译工具版本号。

React Native 作为一个应用开发框架，是依附于 Android 项目的，因此应用名称和应用图标的修改与普通的 Android 项目的修改没有区别。下面简单总结一下。

为了修改应用名称，打开“项目目录”/android/app/src/main/res/values/strings.xml 文件。未修改的文件内容如下：

```
<resources>
  <string name="app_name">Project19</string>
</resources>
```

将其中的 Project19（或者初始化时使用的其他名称）改为所希望的应用名称。

为了修改应用图标，开发者先要准备好 5 个内容相同、分辨率不同的图标，PNG 格式，分辨率是：

- MDPI（Medium Density Screen, 160 DPI），其图标大小为 48px × 48px；
- HDPI（High Density Screen, 240 DPI），其图标大小为 72px × 72px；

- xhdpi (Extra-high density screen, 320 DPI), 其图标大小为 96px × 96px;
- xxhdpi (xx-high density screen, 480 DPI), 其图标大小为 144px × 144px;
- xxxhdpi (xxx-high density screen, 480 DPI), 其图标大小为 192px × 192px。

在“项目目录”/android/app/src/main/res/下有 5 个子目录, 名称分别是 mipmap-hdpi、mipmap-mdpi、mipmap-xhdpi、mipmap-xxhdpi、mipmap-xxxhdpi。每个子目录下都有一个名为 ic_launcher.png 的文件, 用相应分辨率的新图标文件替换 ic_launcher.png 这个文件。

提示: 有可能没有 mipmap-xxxhdpi 文件夹, 请开发者自行建立并复制文件。

为了修改项目编译的 SDK 版本号和编译工具版本号, 使用 Android Studio 打开项目, 在主菜单中选择“File”, 然后选择“Project Structure”子菜单项, 出现如图 18-10 所示的对话框。选择左侧最下方的“app”, 然后在右侧就可以修改项目编译的 SDK 版本号和编译工具版本号了。

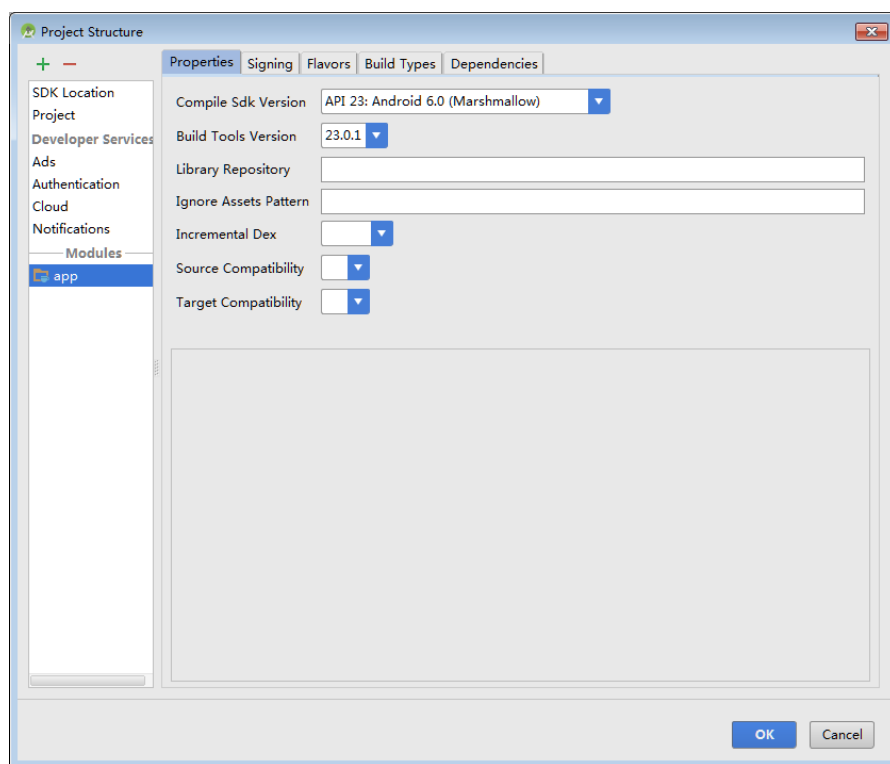


图 18-10 修改项目编译的 SDK 版本号和编译工具版本号

React Native 初始化项目在 Android 平台下默认是显示方式跟随设备放置方式而改变。如果希望移动应用的显示方向始终不变, 则需要修改 React Native 项目目录下的\android\app\src\main\AndroidManifest.xml 文件, 为.MainActivity 加入 android:screenOrientation 属性。修改后的内容如下:

```
.....
.....
<activity android:name=".MainActivity"
    android:label="@string/app_name"
    android:configChanges=
```

```
"keyboard|keyboardHidden|orientation|screenSize"
android:screenOrientation="portrait"> //新增属性
```

.....

其中，`android:screenOrientation="portrait"` 表示程序始终竖屏显示，而 `android:screenOrientation="Landscape"` 表示程序始终横屏显示。

18.4 Android 平台应用生成发布版本安装包

Android 应用在发布前，需要生成发布版本安装包，然后上传至发布平台。

18.4.1 生成发布密钥

在生成发布版本安装包前，需要先生成发布密钥。Android 开发环境自带了密钥生成工具。如果开发者没有发布密钥，则在命令行窗口输入以下命令生成一个发布密钥：

```
keytool -genkey -v -keystore my-release-key.keystore -alias my-key-alias -keyalg RSA
-keysize 2048 -validity 10000
```

命令中的 `my-release-key.keystore` 是生成的密钥库文件名称，开发者可以修改这个字符串为自己所希望的名称。`my-key-alias` 是生成的密钥库别名，开发者也可以将它改成自己所希望的名称。

回车后，命令行窗口会提示输入密钥库（keystore）和对应密钥的密码，还有名称、公司名称、公司简称等信息。最后会在当前目录下生成一个指定名称的密钥库文件。

密钥文件生成后，将它移动到项目目录的 `android/app` 子目录下。

18.4.2 修改 gradle 配置文件

为了生成发布版本，需要修改 React Native 项目中 Android 项目的 `build.gradle` 文件。这个文件位于项目目录的 `android/app` 子目录下，将其内容修改为（需要添加的内容都已经做了注释）：

```
.....
defaultConfig {
    applicationId "com.project19"
    minSdkVersion 16
    targetSdkVersion 22
    versionCode 1
    versionName "1.0"
    ndk {
        abiFilters "armeabi-v7a", "x86"
    }
}
signingConfigs {
    //signingConfigs 整个段落都需要添加
    release {
        storeFile file("../my-release-key.keystore") //使用刚才生成的密钥文件名称
        storePassword "123456" //使用 18.4.1 节中生成密钥时输入的密码
        keyAlias "my-key-alias" //使用 18.4.1 节中生成密钥时输入的别名
        keyPassword "123456" //使用 18.4.1 节中生成密钥时输入的别名密码
    }
}
```

```

    }
  }
  buildTypes {
    release {
      minifyEnabled enableProguardInReleaseBuilds
      proguardFiles getDefaultProguardFile("proguard-android.txt"), "proguard-rules.pro"
      signingConfig signingConfigs.release //这句话需要加上
    }
  }
  // applicationVariants are e.g. debug, release
  applicationVariants.all { variant ->
    .....
  }
}

```

18.4.3 生成发布版本安装包

在命令行状态下，进入项目目录的 `android` 子目录下，输入命令：

```
./gradlew assembleRelease
```

回车执行，就会生成发布版本安装包。它的位置是：“项目目录”/android/app/build/outputs/apk/app-release.apk。

18.5 其他组件与 API

在 React Native 开发中，还有一些组件和 API 本书并没有详细讨论，在此略作总结。

18.5.1 动画相关

在移动应用中，精美的动画能让用户体验更好。在 React Native 框架中，与动画相关的 API 目前还处于调研开发阶段。从发展趋势来看，动画功能将主要由两个互补的 API 完成。它们是：

- `LayoutAnimation` API，可以让某容器内所有组件一起产生动画效果。
- `Animated` API，让某个组件具有可交互控制的动画效果。

在前面的章节中我们已经看到，当改变组件的某些属性、样式值时，组件会发生相应的外观变化。通过定时器等机制让这种变化每秒发生 30 次，就是一种简单、有效的动画实现手段。

又如第 11 章所示，开发者可以控制 UI 组件跟随手势产生视觉上的变化，这也是一种简单、有效的动画实现手段。

更精美的动画设计实现是专业性非常强的分支，通常由专业人员来完成。目前 React Native 的动画相关功能还不足以让动画设计专业人员放弃原来的动画实现手段，而采用 React Native 框架提供的动画功能来实现。使用 React Native 开发的移动应用在使用动画效果时，更推荐使用混合开发方式来实现。

18.5.2 其他未讨论的组件与 API

还有一些组件和 API, 如 `DrawerLayoutAndroid`、`PullToRefreshViewAndroid`、`SegmentedControlIOS`、`TabBarIOS`、`ViewPagerAndroid`, 没有讨论它们的原因是: 这些组件和 API 不是跨平台的, 功能也没有太多的亮点, 或者可以由开发者自己利用 React Native 代码写出一个类似的跨平台实现。

React Native 0.21.0 版本引入了一个新组件 `NavigationExperimental`, 从它的名字就可以知道这是个实验性组件, 本书暂不讨论。它将来会用来替代 Navigator API。

附录 A

ECMAScript 2015 语法参考

这部分讨论了 React Native 开发中需要使用到的 JavaScript 某些高级语法，这些高级语法大部分是（但不全部是）ECMAScript 2015 新增的语法，目的是让读者对 React Native 开发中需要使用到的 JavaScript 知识点进行理解足够进行 React Native 开发。

A.1 解构赋值

ES 6 允许按照一定模式，从数组和对象中提取值，声明变量并对其进行赋值。这种语法被称为解构赋值（Destructuring）。

在解构赋值语法出现之前，只能一次声明一个变量并为它赋值，例如：

```
var a = 1;
var b = 2;
var c = 3;
```

使用解构赋值语法，上面的三条语句可以改写为一条语句：

```
var [a, b, c] = [1, 2, 3];
```

解构赋值语法不仅适用于数组，还适用于对象。请看下面的示例代码：

```
let {objA, objC, objB} =
  {
    objA: 'aaa',
    objB: {
      property1: '111',
      property2: '222'
    }
  };
console.log(objA);           //打印出 aaa
console.log(objB.property2); //打印出 222
console.log(objC);           //打印出 undefined
```

对象的解构赋值比数组的解构赋值更灵活，它可以在一条语句中解构出不同类型的变量，而数组只能解构出相同类型的变量。

在数组的解构赋值中，变量的取值由其位置决定。

在对象的解构赋值中，变量的取值与其位置没有任何关系，变量必须与属性同名，才能取到

正确的值。比如上例中的 objC 就没能取到值。

可以在函数参数中使用解构赋值，比如下例：

```
function getFullName( aPerson ) {
    return aPerson.giveName + '.' + aPerson.familyName;
}
```

可以改写为：

```
function getFullName( {giveName, familyName} ) {
    return giveName + '.' + familyName;
}
```

A.2 箭头函数

ES 6 提供了使用“箭头”（=>）符号定义函数的语法。

ES 6 鼓励用箭头符号在需要回调函数的地方直接定义不需要名称、代码简短的函数。这种方式使代码更简洁。

定义箭头函数的示例代码如下：

```
var function1 = para1 => para1;
var function2 = (para1, para2) => para1+para2;    //当参数多于一个时，使用括号
var function3 = ()=>'I do not need para';        //当不需要参数时，使用括号
```

上面的三条语句与下面的代码等价：

```
var func1 = function(para1) {
    return para1;
};
var func2 = function(para1, para2) {
    return para1+para2;
};
var func3 = function() {
    return 'I do not need para';
};
```

如果箭头函数的代码体部分多于一条语句，就要使用大括号将它们括起来，并且使用 return 语句返回。如果箭头函数需要返回一个对象，则必须在对象外面加上括号。这是因为大括号会被解释为代码块。请参见下例：

```
var getTempObj = id => (
    if ( id === undefined) return {
        id: 'not ready',
        name: "tempName"
    };
    else return {
        id: id;
        name: 'tempName'
    };
);
```


A.3 for in 循环语句

JavaScript 允许开发者通过 for in 循环语句遍历数组。它的用法与普通的 for 循环的区别见下面的示例代码。

```
.....
var aArray = new Array();
aArray [0] = "苹果";
aArray [1] = "香蕉";
for(var i = 0; i< aArray.length;i++)
{
    console.log(aArray [i]);    // 先打印苹果, 再打印香蕉
}
for(var key in aArray)          //key 拿到的是下标(将普通数组视为 JavaScript 的关联数组)
{
    console.log (key);          // 打印出 0,1
    console.log(aArray[key]);    // 先打印苹果, 再打印香蕉
}

var dicArray = new Array();      //建立关联数组
dicArray["first"] = "苹果";
dicArray["second"] = "香蕉";
for(var i = 0; i<dicArray.length;i++)    //普通的 for 循环机制无法取到
{
    Console.log(dicArray[i]);
}
for(var key in dicArray)          // 拿到的是关联数组的下标
{
    console.log (key);            // 打印出 first,second
    console.log(dicArray [key]);    // 先打印苹果, 再打印香蕉
}
.....
```

A.4 JSX 的延展属性 (Spread Attributes)

... 操作符 (也被称作延展操作符, Spread Operator) 已经被 ES 6 数组操作支持。相关的还有 ES 7 规范草案中的 Object 的剩余和延展属性 (Rest and Spread Properties)。React Native 利用了这些还在制定标准中就已经被支持的特性来使 JSX 拥有更优雅的语法——延展属性。

请看下面的示例代码:

```
.....
var props = {};
props.propA = x;
props.propB = y;
var component = <Component {...props} />;
.....
```

现在, component 这个 JSX 元素有了 props 变量的所有属性。

A.5 Promise 机制

Promise 机制代表着在 JavaScript 程序中下一个伟大的范式，但是理解其为什么如此伟大不是一件简单的事情。它的核心就是一个 Promise 代表一个任务结果，这个任务有可能完成、有可能没完成。Promise 模式唯一需要的一个接口是调用 then 方法，用来注册当 Promise 完成或者失败时调用的回调函数。

当开发者希望进行一个时间比较长的操作时，也许会将这个操作设计成异步操作。也就是 JavaScript 代码调用了这个操作后，会继续向下执行，而异步操作的执行结果通过调用异步操作时指定的回调函数来通知开发者。通常异步操作的执行结果都有两个：成功或者失败。

在 Promise 机制没有推出之前，开发者通常会这么写：

```
onSuccessCallback(para) {
  .....//处理操作成功事件，para 中是操作成功上传的数据
}
onErrorCallBack(para) {
  .....//处理操作失败事件，para 是操作失败上传的数据
}
.....
try{
  .....
  this.anAsyncFunction( para, this.onSuccessCallback, this.onErrorCallback);
  .....
} catch(error) {
  .....
}
.....
```

使用 Promise 机制后，相应的代码通常被改写为：

```
.....
this.anAsyncFunction( para ).then(
  (para)=> {
    .....//处理操作成功事件，para 是操作成功上传的数据
  }
).catch( (error)=> {
  .....//操作失败与操作中抛出异常都被集中在这里处理
}
)
.....
```

Promise 的真正强大之处在于可以方便地实现 Promise 的多重链接。在官方文档的例子中，下面的代码示范了一个两重 Promise 链接。

```
.....
fetch(REQUEST_URL)
  .then((response) => response.json())
  .then((responseData) => {
    this.setState({
      movies: responseData.movies,
    });
  })
  .done();
.....
```

对 Promise 多重链接的使用，大家有个印象就可以了，当真正需要使用时，很自然地就会使用了。

A.6 数组的 map 成员函数

数组的 map 成员函数是 ES 5 中新加入的特性。map 函数名是“映射”的意思。它的原型是：

```
array.map(callback,[ thisObject]);
```

callback 的参数类似于：

```
callback:function(value, index, array) {
    // ...
};
```

map 函数的作用是按照某种关系从原数组“映射”出新数组。下面的例子是数值项求平方：

```
var data = [1, 2, 3, 4];
var newArray = data.map((item)=>{return item * item});
console.log( newArray );    //输出[1, 4, 9, 16]
```

A.7 ES 6 语法中的类与继承

class 是 ES 6 引入的最重要特性之一。在没有 class 之前，开发者尝试过很多方法来模拟类，但效果都不好。

在 ES 6 中，类的定义方法类似于：

```
class People {
    constructor(name) {          //构造函数
        this.name = name;        //所有的成员变量必须在构造函数中声明
    }
    sayName() {                  //声明并定义成员变量
        console.log(this.name);
    }
}
```

类被定义完成后，就可以按如下方式使用它：

```
var p = new People("Tom");
p.sayName();
```

就像函数有函数声明和函数表达式两种定义方式一样，类也可以通过类表达式来定义：

```
let People = class {
    constructor(name) { //构造函数
        this.name = name;
    }
    sayName() {
        console.log(this.name);
    }
};
```

你可能以为类声明和类表达式的区别在于变量的提升不同。但事实是，无论是使用类声明还

是类表达式的方式来定义，都不会有变量的提升。所以下面的写法是错误的：

```
var p = new People("Tom");    //错误，People 未定义
class People {
    //...
};
```

类中的所有方法默认都是严格模式（strict mode），所以不用再次声明了。

ES 6 支持通过关键字 `extends` 来继承一个类，并且可以通过 `super` 关键字来调用父类的成员变量和成员函数。

```
class Student extends People {
    constructor(name, grade) { //构造函数
        super(name);          //调用父类构造函数
        this.grade = grade;
    }
    sayGrade() {
        console.log(this.grade);
    }
}
```

上面的例子中我们定义了一个 `Student`，它是 `People` 的子类。

A.8 三元运算符

三元运算符也称作三目运算符，它的形式是：

`A? B:C`

它的运算规则是：A 为非零吗（或者 A 为 `true` 吗）？如果 A 非零（为 `true`），则整个表达式的值为 B，否则为 C。

在 JavaScript 中三目运算符有更灵活的用法，请看下面语句：

```
$('#item')[ flag ? 'addClass' : 'removeClass']('className')
```

当 `flag` 值非零（为 `true`）时，就变成以下代码：

```
$('#item')['addClass']('className')
```

当 `flag` 值为零（为 `false`）时，就变成以下代码：

```
$('#item')['removeClass']('className')
```

这种灵活用活，也可以在本书 7.5.3 节的代码 7-18 中看到。